Máster en Ingeniería Informática, Facultad de Informática,
Universidad Complutense de Madrid

UNIVERSIDAD
COMPLUTENSE
MADRID

Ludificación de rutinas de pacientes de oncología usando posicionamiento interno mediante WIFI

Nota: 9 (Sobresaliente)
Convocatoria: Septiembre 2018

Autor: Sergio Moreno de Pradas
Director: Jose Luis Vazquez Poletti
Curso académico: 2017 / 2018

Le dedico este proyecto a todas las personas que siempre han estado a mi lado, que nunca me han dado por perdido y que siempre me han ayudado con todo lo que necesitaba;
a mi tutor, Jose Luis Vazquez Poletti, sin el cual nada de esto habría sido posible y, sobre todo,
a todos los pacientes a los que va a ayudar a sacar una sonrisa.

El/la abajo firmante, matriculado/a en el Máster en Ingeniería Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Ludificación de rutinas de pacientes de oncología usando posicionamiento interno mediante WIFI", realizado durante el curso académico 2017-2018 bajo la dirección de Jose Luis Vazquez Poletti en el Departamento de Arquitectura, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Resumen

El sistema creado como resultado del trabajo de fin de máster presentado consiste en un sistema capaz de añadir ludificación (convertir en un videojuego) a las rutinas que tienen que seguir los niños pacientes de la sección de oncología intentando hacer más amena la estancia de estos y permitir que se abstraigan lo máximo posible. Para conseguir esto se creado un sistema donde los usuarios administrativos puedan crear las rutinas que tienen que seguir los niños (dividido en tareas) de forma diaria, una vez creada la rutina, el sistema generará una lista de misiones que se le encargarán a completar al paciente mediante el uso de una aplicación móvil.

Para mejorar la experiencia de usuario y dar un valor aún mayor al sistema, se ha planteado el uso de geoposicionamiento interno en el hospital mediante el WIFI interno que tiene este y mapas subidos al gestor interno que tenga (por ejemplo en la tecnología CISCO sería el Prime Infrastructure), de esta forma también servirá de guía a los pacientes para moverse por el hospital además de permitir a los administrativos mantener un control sobre la posición de los pacientes.

En resumen, el sistema permite a los pacientes acceder a una aplicación móvil para completar misiones que le reportan recompensas (experiencia, objetos, etc) mientras que el administrativo tiene acceso a una aplicación web para gestionar y mantener el sistema.

Summary

The system implemented as a result of the final master degree project is a system that able to add gamification to oncology patients daily routines trying to liven up the living in the hospital and allowing them to forget about it as much as possible. To allow all of this the system created enables the administratives users from the hospital to create the daily routines the have to follow using tasks and, when it's created, the system turns it to missions the patients have to complete using a mobile app.

To enhance the user experiences and giving even more value to the system, the use of indoor positioning was proposed using the WIFI deployed in the hospital and the maps uploaded to the internal manager (for example, if the manufacturer is CISCO, the manager is the Prime Infrastructure) which not only enables the new patients to move around the hospital but also enables the administratives to keep track of the patients whereabouts.

To sum it up, the system allows the patients access to a mobile app to complete missions that report different rewards while the administrative has access to a web app to manage and maintain the system.

Palabras clave / Keywords

Palabras clave	Key words
Geoposicionamiento interno	Indoor positioning
Ludificación	Gamification
Microservicios	Microservices
WIFI	WIFI
CMX	CMX
Docker	Docker
Kubernetes	Kubernetes
Aplicación móvil	Mobile app
Bots	Bot
Spring	Spring
IONIC	IONIC
Angular	Angular
Node.js	Node.js

Tabla 1: Palabras clave / Keywords

Introducción

El siguiente documento plantea una descripción completa del trabajo realizado para llevar a cabo este sistema, la arquitectura implementada, tecnologías usadas y medidas tomadas para conseguirlo.

El problema a resolver por este proyecto consiste en ser capaz, de alguna forma, de permitir a los pacientes de oncología una capa de abstracción de la realidad que es estar ingresado en esta sección; para esto, se planteó la creación de un sistema que fuera capaz de convertir, en un juego, las rutinas que tienen que seguir los pacientes para transformar las tareas cotidianas en misiones a completar y así conseguir recompensas en el juego. Dado la naturaleza del proyecto, se decidió que el sistema iría enfocado a niños de entre 7 a 14 años que es la edad en la que pueden estar interesados en probar este tipo de aplicaciones teniendo en cuenta el ámbito de la misma y el alcance que puede tener.

Este sistema planteado debía ser capaz de una serie de tareas específicas:

- **Gestión de recursos:** Debía tener una capa de gestión que permitiera, a un usuario administrativo, realizar operaciones de creación, mantenimiento y consulta sobre los diferentes recursos de la aplicación, además, esta gestión debía mostrarse al usuario mediante una interfaz intuitiva que le facilitara la tarea lo máximo posible.

- **Perfiles de personaje:** Los pacientes debían ser capaces de crear un perfil propio donde se describiera el personaje del mismo. El personaje debía tener un alias, clase, nivel, etc de forma que la inmersión en el juego y el deseo de obtener recompensas por cumplir misiones fuera lo más alto posible de forma que el paciente quisiera seguir usándolo.

- **Posicionamiento interno:** El sistema debía ser capaz de posicionar en el mapa del hospital a los diferentes pacientes por dos motivos:

    - Permite guiar al paciente a las diferentes misiones que tiene asignadas, de forma que permite moverse por el hotel a pacientes nuevos sin miedo y aumenta la realidad del sistema.
    - Permite a los usuarios administrativos mantener un control sobre la posición de los diferentes pacientes y así ser conscientes de si está siguiendo la ruta planeada o no.

- **Generador de misiones:** Debía tener un subsistema que, dado una tarea con una cierta clase, un objetivo, una localización, etc. este fuera capaz de generar una misión para los pacientes de forma que la misión fuera lo más desconocida posible, es decir, que no hubieran realizado dicha misión con anterioridad.

- **Comunicación con administrativos/NPCs:** Teniendo en cuenta en el sistema tenía muchas partes donde requería de intervención de usuarios administrativos para continuar con el flujo normal del mismo, era necesario un sistema tal que permitiera la comunicación bidireccional con estos usuarios.

Con todas estas tareas se llegó a la conclusión que el sistema se dividiría en 4 subsistemas, un Backend, un Frontend, un Bot y una aplicación móvil, los cuales se explicarán más adelante.

Introduction

The following document presents a complete description of the work accomplished to create the system, the architecture implemented, the technologies used and decisions taken to achieve it.

This project aims to offer the oncology patients an abstract layer on being admitted to the hospital in that area. To allow this, a gamification system capable of turning the routine in a game and was proposed, the system would be focused to kids between 7 and 14 years old giving that is the range of age in which the kids could potentially be interested in this kind of games.

The system should be able to accomplish certain tasks:

- **Resource management:** it should have a management layer that allows an administrative to create, maintain and consult the different system's resources, besides, this functionalities had to be display on an intuitive and friendly manner to the user.

- **Character profiles:** the patients should be able to create a new profile which describes the character of said user. The character had to have and alias, role, level, etc enhancing the desire of the user to get rewards and improving the inmersion in the game.

- **Indoor positioning:** the system had to be able to position the various users on the hospital's map for two reasons:

    - Allowing the system to guide the patient to the various missions assigned to him/her, allowing to move to new patients that don't have knowledge about the structure of the hospital, besides, it enhances the reality of the system.
    - Administratives are able to keep track about the position of the various patients, that enables them to be aware if they are following the route.

- **Mission generator:** it had to be able to generate a mission for the patients giving a task with a type, an objective, a localization, etc, said mission should be as random as possible so the missions don't become something predictable or tedious.

- **Communication with NPCs:** keeping in mind that the system had several parts which required intervention from administratives to maintain the normal flow, it required a bidirectional system to communicate between administrative and system.

With all these requirements defined, the conclusion was that the system would be divided in 4 differents subsystems:

1. Backend
2. Frontend
3. Bot
4. Mobile app

Estudio previo

Antes de comenzar con la implementación del sistema, fue necesario un exhaustivo estudio para tomar las decisiones en cuanto a qué tecnologías se iban a usar. A la hora de escoger la tecnología a usar en la creación de una parte del sistema lo que ha primado ha sido, no solo rendimiento alcanzable y escalabilidad a futuro si no también el conocimiento previo que se tenía en el uso de dichas tecnologías; de forma que una de las razones primarias para escoger las tecnologías fue si se había usado antes o no.

- Backend:
  - Java 8
  - Spring
  - MongoDB
- Frontend:
  - IONIC
  - Angular
  - SASS
- Bot:
  - Node.js
- Aplicación móvil:
  - IONIC
  - Angular
  - SASS

Visita a hospital

Antes de comenzar con la etapa de diseño, se realizó una visita al hospital **La Paz**, concretamente al ala de Oncología para poder analizar la estructura del edificio, las tecnologías que usaban y que podían ser útiles para el proyecto, estudiar las rutinas diarias que mantenían los pacientes y poder mantener conversaciones con los trabajadores para que pudieran hablar de las cosas que ellos creían que podían funcionar y cuáles no, como era la rutina normal de un paciente, etc.

En la visita se pudieron aprender varias cosas que sirvieron de utilidad a la hora de diseñar el sistema y definir qué servicios se debían implementar y de qué manera:

- Los pacientes en edad de ir al colegio tienen uno propio en el hospital al que tienen que asistir de 10.30 a 13.30 donde realizan actividades normales de una clase, incluso realizan exámenes y tienen evaluaciones.

- Existen eventos para los pacientes tanto internos del hospital como pueden ser concursos de postales o de dibujos como externos a este como visitas de voluntarios, eventos de empresas, etc.

- La información obtenida de los administrativos ha aclarado las tareas que deben tener como posibilidad para introducir a la hora de crear la rutina para el paciente:

    - Asistencia a colegio.
        - Tanto en su habitación (van los médicos) como en la sala de colegio.
    - Analítica.
    - Visitas de médico.
    - Pruebas médicas (punción, rayos, catéter, cirugía).
        - Tanto en su habitación como en una sala.
    - Dormir.
    - Tomar la medicación.
        - Puede ser a la hora del colegio.
        - Si es quimioterapia pueden ir a su habitación
    - Comida.
    - Eventos externos.
        - Visita de voluntarios.
        - Eventos de empresas como Ubisoft.
        - Teatro.
        - Talleres.
        - Etcétera.
    - Eventos internos.
        - Concursos de diversas categorías.
        - Talleres en el ciberaula.

- El Wifi del hospital lo controla **Juegaterapia,** una organización sin ánimo de lucro con la que se planteó una colaboración. De salir dicha colaboración, facilitaría la tarea de conectar dicha infraestructura con el sistema y permitir el uso del posicionamiento interno, además de permitir realizar la configuración inicial de la misma por los desarrolladores que son conscientes de la información que necesitan.

- En la conversación mantenida con los profesores del colegio se planteó, por petición de estos, la inclusión de varias funcionalidades que se dejaron como *features* a incluir siempre que diera tiempo a ello una vez se estuviera creada la base del sistema. Las funcionalidades son:

    - Poder avisar al profesor por un mensaje cuántos alumnos se esperan en clase ese día.
    - Permitir al docente convertir las varias tareas de la clase (como exámenes) en retos y misiones.
    - Permitir evaluar por rendimiento a los alumnos y, en base a la puntuación que obtengan, otorgar recompensas en el juego.
    - Añadir retos en **Scratch** para los alumnos.

    - Incluir la opción de jugar diferentes juegos (se dió el ejemplo de juegos de mesa fácilmente transformables a juegos digitales como **Lobo**).
    - Permitir al paciente leer ciertos libros que se aconsejan y recompensar en función de lo que lea.

- En la conversación mantenida con las enfermeras plantearon la opción de incluir el uso de **Kahoot!** en la aplicación, una plataforma gratuita muy usada en la actualidad que permite la creación de cuestionarios donde pueden entrar una gran cantidad de usuarios al mismo tiempo para competir, incluso permite la creación de grupos.

Con toda esta información, se diseñó el sistema intentando crear una base que permitiera crear todas las funcionalidades pedidas o, de no poderse implementar por tiempo, dejarlo para facilitar lo máximo posible su inclusión a futuro.

Arquitectura del sistema

El sistema está dividido en cuatro partes fundamentales, algunas las cuales, a su vez, están subdivididas en diferentes subsistemas con arquitecturas propias que se detallarán más adelante en el documento. Las cuatro divisiones principales son:

- **Backend:** El backend del sistema tiene implementada toda la lógica relacionada con **usuarios web, usuarios de aplicación móvil, rutinas y la geolocalización.** El backend se conectará con las demás partes del sistema cuando sea necesario enviarles información relevante para estos y avisar de un acontecimiento importante, además, permitirá que los demás se conecten con él para realizar ciertos servicios como obtener información en un momento dado o modificar el estado de base de datos.

- **Frontend:** El frontend es la parte del sistema que permite al usuario la interacción con las diferentes funcionalidades que ofrece el backend, este usuario será de tipo administrativo ya que será el que se encargue de la gestión y mantenimiento de los recursos del sistema (usuarios web/administrativo, usuarios aplicación móvil, rutinas). Para acceder a esta parte del sistema el usuario debe pasar una pantalla de autenticación que, usando sus credenciales, pueda indicar al sistema que se trata de un usuario con capacidad para ingresar en esta parte; también se tiene una parte de autorización que, dependiendo de la funcionalidad que se quiera realizar y los permisos que tenga el usuario se le permitirá o denegará el acceso.

- **App. Móvil:** La aplicación móvil se usa por los usuarios finales que serán los que vayan realizando las misiones en las que consiste su rutina, es decir, los pacientes de oncología que tengan disponible la aplicación. Esta parte del sistema se comunicará con el Backend para obtener la lista de misiones de las que se compone la rutina del usuario/paciente y se la mostrará para que sea consciente de las tareas que tiene que realizar hoy, además, será capaz de guiar al paciente por el hospital para completar dichas misiones permitiendo también a los usuarios administrativos el saber la posición del paciente en caso de necesitarla.

- **Bot:** La última parte del sistema se trata de un bot que es el encargado de la comunicación con los usuarios NPC (usuarios administrativos que han elegido actuar como esto) para hacerle saber si tiene alguna misión de un usuario de aplicación móvil asignada, también permite al usuario administrativo hacer saber al Backend si se ha cumplido una misión concreta que requería de un *feedback* por parte del este para marcar como completa (ha entregado un objeto, ha completado un acertijo, …).

Todas estas partes de la aplicación tendrán una arquitectura propia con tecnologías concretas usadas en la implementación de las mismas, todo esto se explicará más adelante de forma más detallada en el apartado de cada división.

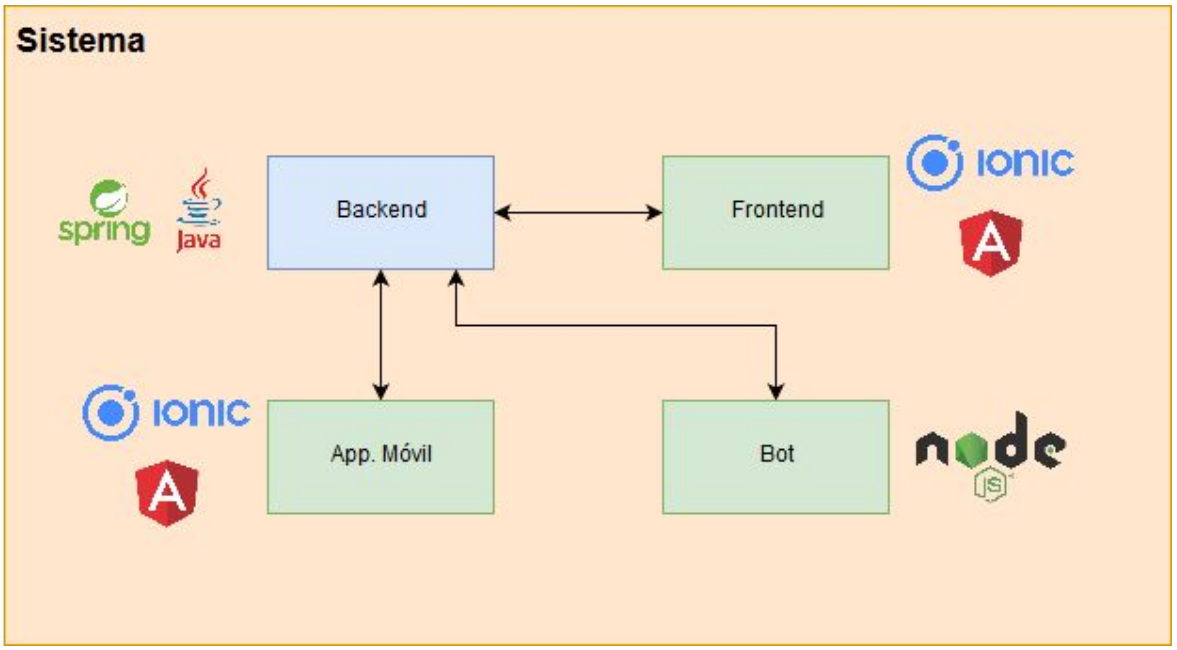

Figura 1: Diagrama de arquitectura del sistema con tecnologías.

Backend

Tecnologías empleadas

Para la creación del Backend se ha utilizado diferentes tecnologías, cada una teniendo un propósito diferente:

- **Java 8** → Lenguaje de programación en el que se ha implementado el Backend, una de las razones de usar Java como base es debido a la existencia del Framework de desarrollo **Spring** que se explicará más abajo y que permite implementar una gran cantidad de funcionalidad mediante poco código debido al uso de anotaciones propias del framework, facilitando así la tarea de desarrollo.

  Otra razón para su uso es que, al requerir en todo momento la creación de clases, interfaces, colecciones de información, etc se consigue un sistema muy bien estructurado (aunque eso aumente el tiempo de desarrollo) lo que facilita el mantenimiento y escalabilidad a futuro.

  Concretamente se ha usado la versión 8 por la integración de funciones de **MapReduce** que facilita el tratamiento sobre grandes colecciones de datos.

Figura 2: Logo de Java.

- **Framework Spring** → El Framework Spring es la tecnología más importante en el desarrollo del Backend. Spring se trata de un framework de desarrollo que permite una gran cantidad de funcionalidades para incluir en los proyectos o facilitar la creación de las mismas, consta de muchas partes pero en el Backend se han usado las siguientes:

    - *Spring Boot:* Proyecto de Spring que permite desplegar de forma automática las aplicaciones Java creadas con este, además, permite también crear Clases Java definidas con la anotación **@Configuration** (que indica que se trata de configuración para la aplicación) o ficheros XML donde se puede definir configuración para aspectos clave, por ejemplo en la configuración de la conexión con base de datos, la configuración de seguridad, etc.

    - *Spring Security:* Toda la configuración de seguridad de la aplicación Backend está definida con Spring Security, entre otras cosas permite:
        - Definir filtros de forma que el acceso a un servicio concreto esté restringido por usuario o rol.

        - Definir la página de login y logout junto con el flujo completo de ejecución de esta, qué hacer antes de autenticarse, una vez autenticado y tras autenticarse (lo mismo al hacer logout).

        - Definición de HTTPS con toda la configuración necesaria, como los certificados.

    - *Spring Framework:* Proyecto que contiene toda la función core del framework Spring, en concreto, en el proyecto Backend se usa para definir toda la estructura del proyecto, los controladores de llamadas HTTP, los servicios que se conectan con los repositorios de base de datos para realizar las funciones, los servicios desplegados con sus parámetros, etc.

- *Spring Data MongoDB:* Spring Data tiene implementado todas las funcionalidades necesarias para conectarse con diferentes bases de datos junto con una interfaz de comunicación con estas para acceder a los datos y realizar operaciones sobre estos, en concreto, el sistema utiliza las funcionalidades que tiene con MongoDB al ser esta la base de datos que se usa en el proyecto, lo que permite definir la relación de las colecciones con los clases de objetos, definir las querys a utilizar en la aplicación, etc.

Figura 3: Logo de Spring.

- **Maven** → Maven es una herramienta de Apache que se usa para la gestión y construcción de proyectos Java permitiendo actuar como un gestor de librerías, de forma que se listan las dependencias necesarias para que el proyecto funcione en un fichero xml denominado **pom.xml** junto con las configuraciones de construcción del proyecto, en el caso del Backend sería desplegarlo en Docker.

```
<groupId>TFM.microservices</groupId>
<artifactId>app-users-microservice</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>app-users-microservice</name>
<description>Microservice for app users in TFM</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-boot-admin.version>2.0.0</spring-boot-admin.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-cache</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

Figura 4: Ejemplo de fichero pom del microservicio de usuarios app. móvil.

Figura 5: Logo de Maven.

- **Swagger** → La herramienta Software de Swagger permite documentar las API REST indicando tanto método HTTP necesario para realizar la llamada como argumentos que deben llevar, el tip ode los mimos, condiciones específicas que tengas, si son obligatorios o no, incluso la parte donde irían en la llamada (path, query o body). Posibilita el añadir una descripción a cada servicio desplegado para que se pueda analizar, de forma rápida y sencilla, qué funcionalidad ofrece cada uno, todo esto mediante una página web **HTML**.

  Lo más potente de Swagger es que permite no solo analizar los servicios por parte de un usuario externo, si no que también se puede realizar pruebas para comprobar si los servicios funcionan, el retorno de cada uno, posibles estados HTTP a devolver y en qué situaciones, etc. Todo esto facilita en gran medida el uso de una API REST por cualquier otro servicio o recurso y aporta un entorno de pruebas manual.

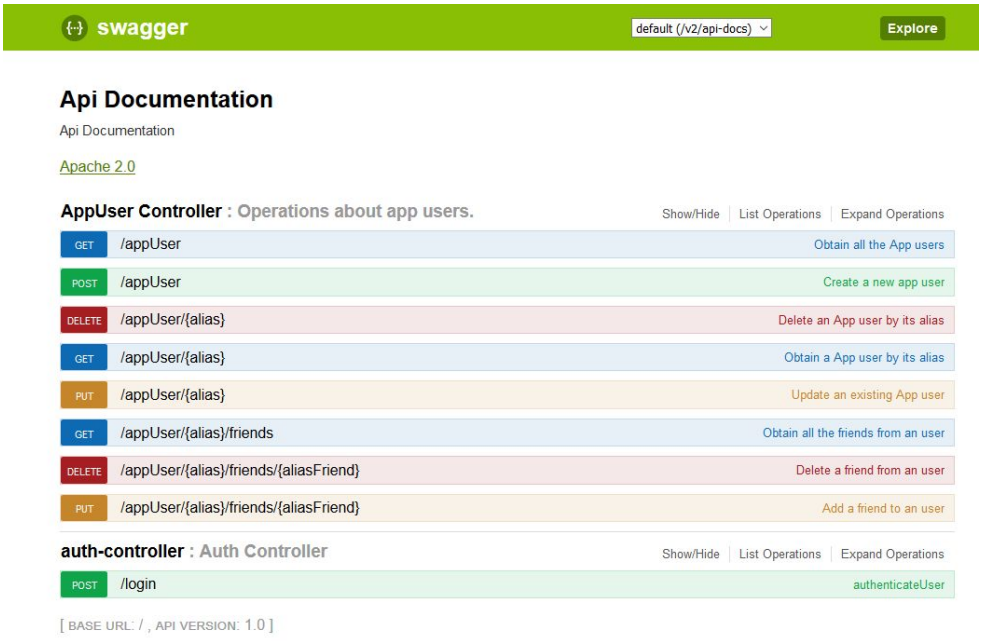

Figura 6: Documentación de la API Rest del microservicio usuarios app. móvil con Swagger.

Figura 7: Logo de Swagger.

- **MongoDB** → La tecnología de base de datos usada es MongoDB, una base de datos orientada a documentos donde, en cada documento, se tienen colecciones. Es un tipo de base de datos noSQL de forma que no se tiene que definir un esquema previo para almacenar en las colecciones (tablas) y, tiene la ventaja, de almacenar los datos en forma muy parecida a JSON lo que facilita el análisis de los datos usando una API Rest donde tanto entrada como salida es en dicho formato.

  El uso de MongoDB viene dado porque es la tecnología de base de datos donde se tiene más experiencia en el desarrollo de aplicaciones lo que agiliza la implementación del Backend al no necesitar un estudio previo de cómo usar dicha tecnología, además de ser una tecnología con alto rendimiento y estar preparada para el tratamiento de grandes colecciones de datos.

Figura 8: Logo de MongoDB.

Arquitectura

El Backend está implementado siguiendo una arquitectura de microservicios separando la funcionalidad del sistema en 4 diferentes:

- **Usuarios web:** Se encargará de toda la gestión asociada con este recurso, tiene funcionalidad CRUD de forma que se pueden crear, obtener, modificar y borrar los usuarios web. También se encargará de añadir el identificador de chat con el bot para cada usuario NPC del sistema.

- **Usuarios app. móvil:** Permite la gestión del recurso de usuarios de la aplicación móvil, no sólo tiene operaciones CRUD sobre este recurso, si no que también se encargará de gestionar las recompensas obtenidas y la gestión de los amigos.

- **Rutinas:** El microservicio de rutinas ofrece operaciones CRUD sobre el recurso de rutinas de la misma forma que los dos anteriores pero, además, tiene la funcionalidad de generar una misión asociada a cada tarea de la rutina; es decir, tiene implementado el generador de textos de misión para los usuarios de app. móvil. También será el encargado de la comunicación bidireccional con el bot y se comunicará con el microservicio de la aplicación móvil para hacerle saber que una misión ha sido completada.

- **Localización:** El microservicio de localización del sistema ofrece las funcionalidades relacionadas con la localización en interiores, estas funcionalidades son tanto obtener la posición concreta de un usuario como obtener las imágenes del mapa donde se va a posicionar en formato *png*.

Cada microservicio está generado como una aplicación Spring Boot teniendo un controlador por recurso que controla, que es el que ofrece los servicios REST para su uso por terceros. Este controlador se comunica con un service que es el que se encarga de la lógica del sistema conectándose con el repositorio de base de datos necesario y realizando las transformaciones convenientes para devolverlo en el servicio siempre como un JSON usando la funcionalidad de Jackson que mapea automáticamente los objetos devueltos a este formato. De forma transversal se tiene en cada microservicio un gestor de excepciones, de forma que se descentraliza esta gestión de cada servicio y, de saltar alguna excepción, se redirigirá automáticamente a este componente para su tratamiento. A continuación se mostrará un diagrama de lo explicado:

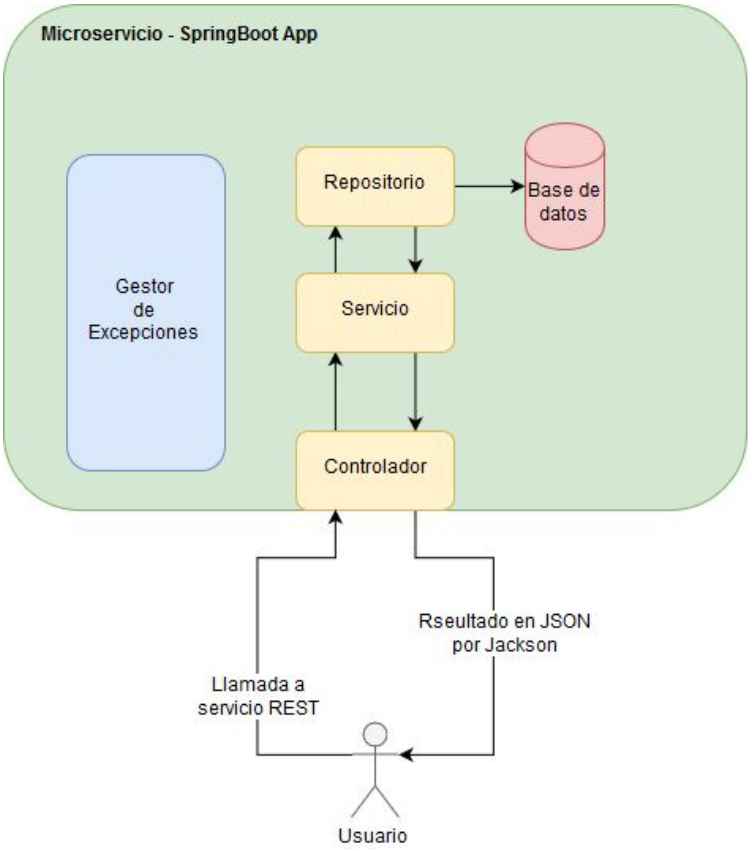

Figura 9: Flujo de microservicio.

Una vez explicado de forma genérica cómo está estructurado cada microservicio, se pasará a definir cómo se ha desarrollado cada parte que lo compone:

- **Controlador:** Para este componente se ha implementado una clase Java que contiene un método por cada servicio HTTP que se ofrece para dicho recurso (GET, POST, DELETE, PUT). Para mapear estos elementos con el framework Spring se han utilizado las anotaciones que este ofrece y son las siguientes:

    - *RestController:* anotación que define una clase como un controlador de Spring de forma que es detectada de forma automática por el escaneo de classpath. Tiene por defecto la anotación de *RequestBody* de forma que mapea los objetos devueltos al body de la request en formato JSON, lo que es muy útil para controladores de API REST.
    - *RequestMapping:* anotación que permite mapear una URL a un método, de forma que cuando se realice una request a la URL especificada se ejecutará el método al que se llamó.
    - *PathVariable:* permite enlazar un dato obtenido por el path de la url a una variable de Java.
    - *RequestBody:* permite transformar el body que está en la request a un objeto definido por el usuario, es decir, transforma el mensaje de formato JSON a un objeto con atributos siendo las claves y teniendo los valores almacenados.

    - *Autowired:* concretamente esta anotación se usó para que Spring generará instancias de un cierto objeto de forma automática cada vez que se cree el objeto que la contiene, por ejemplo cada controlador tiene un atributo que es el servicio al que tiene que llamar y dicho servicio tiene una anotación autowired, de forma que cada vez que se crea una instancia de controlador Spring automáticamente le asociará un servicio.

- **Servicio:** La clase que implementa el servicio es, en definitiva, un controlador de la capa MVC, esto es porque contiene la lógica para realizar todas las tareas definidas por el controlador; cada *endpoint* del controlador tendrá en el servicio un método para resolverlo. En la creación de esta clase se han usado anotaciones como en el controlador para trabajar con Spring:

    - *Autowired:* se utiliza para generar una instancia del repositorio del recurso de forma que se inyecte automáticamente.

    - *Component:* esta anotación permite definir a la clase como candidata para la autodetección que realiza Spring durante el proceso de configuración y el escaneo de classpath. En definitiva marca una clase como un *Bean* de Spring para que al detectarlo lo cargue en el contexto de la aplicación.

- **Repositorio:** La clase del repositorio marcará las consultas que se pueden realizar contra la tabla que tienen enlazada en base de datos. El uso de esta clase permite varias ventajas, la primera es que las consultas son parametrizadas por lo que no existiría vulnerabilidad de inyección de código SQL, otra es que las consultas las crea automáticamente Spring Data MongoDB (en el caso de este sistema al ser esa la base de datos usada), únicamente se tiene que definir ciertos aspectos de la consulta para que este pueda generarla, y eso se consigue con la anotación *Query* del framework.
  Esta anotación permite definir cómo se debe enlazar los argumentos recibidos en el método a la consulta que se va a realizar, como se puede apreciar en la imagen de abajo, el argumento recibido como parámetro *alias* es de tipo String y se tiene mapeado en la anotación Query al primer argumento de la consulta (el primer argumento viene marcado con *?0*); además, también permite definir qué columnas de la colección se van a traer con la consulta usando el atributo *fields*.

```java
public interface AppUserRepository extends MongoRepository<AppUserInputVO, Integer>{

    @Query("{ 'alias' : ?0 }")
    AppUserInputVO findOneByAlias(String alias);

    @Query("{ 'mail' : ?0 }")
    Boolean existsByMail(String mail);

    @Query("{ '_id' : ?0 }")
    AppUserInputVO findOneByUserId(String userId);

    @Query(value="{ 'alias' : ?0}", fields="{ 'friends' : 1}")
    AppUserFriendsStringVO findFriendsByAlias(String alias);

    @Query(value="{ 'alias' : ?0}", fields="{ 'alias' : 1, 'role' : 1, 'level' : 1}")
    FriendVO findFriendInfoByAlias(String alias);
}
```

Figura 10: Ejemplo de repositorio de Base de Datos.

Para que el repositorio funcione correctamente, necesita de un objeto que relacione estas consultas con la tabla de base de datos, además de tener la estructura de dicha tabla (nombre de los atributos, clase de datos que almacenan, …) y para esto es necesario tener una clase Java que representa al documento que se almacene en dicha tabla.

```java
@Document(collection = "app_users")
public class AppUserInputVO {
    @Id
    @JsonIgnore
    private String id;

    private String alias;
    private Integer level;
    private String type_template;
    private String role;
    private List<String> missions;
    private String mail;
    private String phone;
    private String telegramAlias;
    private String position;
    private List<String> friends;
    private String password;
```

Figura 11: Ejemplo de objeto representando un documento.

Como se puede apreciar en la imagen la clase únicamente tiene los atributos que tendrá en la tabla (filas en base de datos relacionales) junto con los *getters* y *setters* de cada uno para que Spring pueda trabajar con este componente.

De nuevo, son necesarias ciertas anotaciones de Spring para que todo funcione correctamente:

    - *Document:* permite relacionar la clase con un el documento y colección de la base de datos de MongoDB, se recuerda que la analogía con bases de datos relacionales sería {colección → tabla, documento → fila de tabla} de forma que al poner la anotación en la clase se indica que la estructura del objeto será la que tenga los documentos y, usando el atributo *collection* podemos indicar la colección a la que se va a conectar. Se puede ver un ejemplo en la imagen de arriba.

    - *Id:* anotación para indicar que el atributo hace referencia al ID de la tabla en base de datos.

    - *JsonIgnore:* permite definir un atributo como ingorado en el JSON generado a partir de dicha entidad, es decir, que cuando Jackson transforme la instancia a un JSON para devolver en la response ignorará estos campos. Un ejemplo sería en las contraseñas de los usuarios que no se deberían devolver.

- **Gestor de excepciones:** Para la gestión de excepciones se ha usado una anotación añadida en Spring 3 llamada *ControllerAdvice*. Dicha anotación define una clase como un gestor de excepciones de forma global, por lo que al saltar una excepción en cualquier parte del sistema, se ejecutará el método definido para tratar la excepción en esta clase. Para enlazar los métodos de la clase a las excepciones que tienen que tratar se usa la anotación *HandleException* donde se lista todas las excepciones de las que se encargará. A continuación se muestra una imagen de un ejemplo.

Estructura de base de datos

Microservicio de usuarios web

Los usuarios web de la aplicación están almacenados en una colección de base de datos llamada *web_users*, dicha colección tiene una serie de campos donde se tiene guardada la información relevante para cada usuario administrativo del sistema. La información es la siguiente:

- **_id** → Cadena de caracteres generada de forma automática que representa de manera única al usuario, se utiliza en las llamadas para obtener información, actualizar o borrar a dicho usuario para evitar problemas de varios resultados al realizar estas operaciones. Se ha mantenido la generación automática dado que la generación es pseudoaleatoria y no proporciona información relevante sobre el usuario, lo que ayuda a la anonimización de los datos.

- **Nombre y apellidos** → los nombres y apellidos del usuario para mostrar a los administrativos y que puedan relacionar al usuario con una persona, de esta forma sabrán de quién se está hablando.

- **Dirección de correo** → información de contacto en caso de que se necesite enviarle un correo o se cree algún servicio mediante el cual el sistema se comunique con los usuarios por correo a futuro.

- **Teléfono móvil** → información de contacto que tiene la misma utilidad que el correo electrónico, únicamente que en vez de ser comunicado mediante correo lo sería mediante un mensaje de texto o, en su defecto, una llamada.

- **Alias de telegram** → el alias que representa al usuario en la aplicación de mensajería Telegram, aplicación que usa el Bot del sistema para comunicarse con los diferentes usuarios administrativos que actúan como NPC y tienen un rol relevante en las misiones. Es necesario para permitir al sistema comunicación con este y obtener información del mismo.

- **Puesto de trabajo** → el puesto de trabajo que tiene el usuario administrativo en el hospital, actualmente solo se tiene de forma informativa aunque a futuro puede ser útil para alguna característica del sistema.

- **Último acceso a la aplicación** → únicamente se tiene a nivel de registro de acciones, este campo permite al sistema tener conocimiento de la última vez que dicho usuario se autenticó con éxito.

- **Última acción en la aplicación** → de nuevo, igual que en el último acceso, este campo se tiene únicamente a modo de registro. Permite al sistema conocer la última acción que realizó el usuario cuando se autenticó por última vez.

- **Contraseña** → la contraseña del usuario para poder comprobar que, cuando el usuario se autentica en el sistema, introduce credenciales válidas. Para aumentar la seguridad del sistema y proteger las credenciales de los usuarios en caso de filtración de la base de datos, se ha hecho uso de una Key derivation function (KDF) llamada BCrypt, que permite usar técnicas de *hashing* para mantener las contraseñas seguras e ilegibles en caso de que un tercero obtuviera acceso a esta información.

A parte de la colección que representa a los usuarios de la aplicación, esta base de datos tiene una segunda colección auxiliar que únicamente sirve para un propósito y es el de permitir al bot almacenar los identificadores de chat de los usuarios de Telegram y poder comunicarse con estos mediante mensajes en un futuro. La colección tiene, de igual manera que *web_users* un _id que se genera de forma automática y dos campos más que son los importantes:

- **Alias** → alias de telegram del usuario administrativo al que pertenece el identificador de chat que se va a almacenar, al almacenar el alias aquí se puede relacionar esta colección con la anterior y, de esta forma, conocer el identificador de chat del usuario y comunicarse con el mismo.

- **Identificador de chat** → el identificador de chat del usuario es una parte esencial para la comunicación que mantiene el bot con el usuario, dado que el bot no puede ser el que comience dicha comunicación, únicamente lo puede hacer disponiendo antes de un identificador que represente el chat con el usuario.

A continuación se muestra un diagrama que representa el estado de la base de datos de este microservicio:

Figura 12: Diagrama de la base de datos del microservicio web users.

Microservicio de usuarios de aplicación móvil

El microservicio de los usuarios de aplicación móvil tiene una base de datos con una única colección en esta y es la que almacena la información relevante a los usuarios (pacientes) que están registrados. La colección se llama *app_users* y dispone de la siguiente información:

- **_id** → de igual manera que en las colecciones del microservicio de usuarios web, el identificador es generado de manera automática y representa de manera única al usuario.

- **Alias** → alias que escoge el usuario de la aplicación para hacerse llamar así en el sistema, representa el nombre de usuario que tiene el paciente y el nombre del personaje que decide ser.

- **Nivel** → nivel que tiene actualmente el usuario en el sistema, comienza con el nivel 1 y, actualmente, no tiene un máximo de nivel. La experiencia necesaria para subir de nivel se calcula en base al nivel actual y ciertos datos que, usando una función matemática, lo calculan.

- **Experiencia actual** → la experiencia de la que dispone ahora mismo el usuario, de esta forma se puede saber cuando tiene que subir de nivel al completar misiones.

- **Plantilla escogida** → la plantilla hace referencia al tipo de vocabulario escogido por el usuario, el usuario de aplicación móvil puede escoger si el vocabulario de las misiones es de tipo pirata, astronauta, normal, etc. Se definen diferentes plantillas para cada uno y se genera el texto de la misión acorde a esto.

- **Rol escogido** → el rol escogido por el usuario es el tipo de personaje que quiere tener en la aplicación, puede ser mago, guerrero, los tipos normales de personaje que se tienen en los videojuegos actuales.
- **Correo electrónico** → el correo electrónico únicamente se tiene de forma informativa y por si fuera necesario ponerse en contacto con el usuario por correo.

- **Teléfono móvil** → de forma análoga al correo electrónico, esta información únicamente se tiene por si fuera necesario ponerse en contacto con el usuario mediante mensaje de texto o llamada.

- **Alias de telegram** → el alias del usuario en Telegram se tiene para permitir a los amigos conectar entre ellos y mantener conversaciones mediante esta plataforma de mensajería al mostrar los alias que tienen los usuarios de la lista de amigos de cada uno y, de esta forma, les proporciona la información necesaria para iniciar un chat con ellos.

- **Amigos** → la lista de usuarios que son amigos del usuario del que se está almacenando la información.

- **Contraseña** → de igual forma que en la aplicación web, la contraseña está almacenada usando BCrypt para protegerla en caso de filtración de base de datos.

Microservicio de rutinas

El microservicio de rutinas tiene la estructura de base de datos más compleja de todas, en total consta de cinco colecciones donde se almacena toda la información relevante a las rutinas de los diferentes usuarios de la aplicación móvil. Las colecciones son las siguientes:

- Routines
  La colección de *routines* almacena la rutina creada por el usuario administrativo para un paciente concreto, tiene almacenado la iformación relevante de esta para, posteriormente, generar las misiones para cada una de las tareas que conforman la rutina. La información que tiene almacenada es:

    - _id → identificador único para la rutina formado por el sistema, tiene el formato:
      {alias del paciente}_{día de la rutina}_{mes de la rutina}_{año de la rutina}

    - Alias del usuario → alias que tiene en el sistema el paciente para el que se está creando la rutina.

    - Creador → correo del creador de la rutina.

    - Fecha de la rutina → fecha para la que, el paciente, tendrá que realizar la rutina creada.

    - Tareas → lista de tareas que forman la rutina, cada tarea tiene un tipo de tarea (colegio, evento, etc), una fecha de inicio y de final, un identificador de NPC si tiene uno seleccionado y un identificador de habitación siempre que sea en una la misión.

- Task_maps
  La colección de *task_maps* contiene la relación entre los tipos de tareas que se definen en la creación de rutinas y los tipos de misión que pueden tener (infiltración, rescate, etc). La información que tiene almacenada en cada documento es la siguiente:

    - _id

    - Tipo de tarea → el tipo de tarea para el que se va a definir la lista de misiones posibles en que se puede convertir.

    - Lista de tipo de misiones → lista de misiones que pueden generarse a partir del tipo de tarea.

- Templates
  Templates es la colección que contiene la información necesaria para crear los textos de misión, para cada tipo de misión tiene una lista de los campos que tiene que tener el texto y, de esta forma, permite generarlos de forma dinámica y lo más aleatoria posible. La información que contiene cada documento es la siguiente:

    - _id

    - Tipo de misión → el tipo de misión para el que se va a relacionar la lista de campos en el texto.

    - Lista de campos de texto que conforman el texto de la misión.

- Mission_options
  La colección de *mission_options* contiene las opciones posibles para cada campo del texto a generar para una misión concreta. En función del tipo de misión y de la plantilla escogida por el paciente al que se le va a enviar la misión se obtendrá una serie de opciones para cada campo, de entre los que se escogerá de forma pseudoaleatoria. La información almacenada es la siguiente:

    - _id

    - Tipo de misión a la que hace referencia.

    - Campo concreto de los necesarios para crear el texto de la misión de dicho tipo.

    - Nombre de la plantilla de las disponibles a usuario a la que hace referencia las opciones.

    - Lista de opciones posibles para dicho campo en ese tipo concreto de misión y usando la plantilla de usuario escogida.

- Missions_routines
  La última colección de la base de datos de rutinas es la que tiene almacenadas las misiones generadas por el generador de misiones en base a las rutinas creadas por los administrativos. Será la información que se muestre directamente al usuario de la aplicación móvil cuando este liste las misiones que tiene para dicho día, la información que se tiene almacenada en cada documento es la siguiente:

    - _id

    - Identificador de la rutina de paciente a partir de la cual se ha generado esta lista de misiones a cumplir.

    - Lista de misiones a cumplir por el paciente → la lista de misiones que deberá cumplir en dicho día para cada cual se tiene almacenada la siguiente información:

        - Texto de la misión que se ha generado de forma pseudoaleatoria.
        - Estado de la misión (Cumplida, no cumplida).
        - Tipo de misión.
        - Fecha de inicio de la misión.
        - Fecha de final de misión.

A continuación se muestra un diagrama que muestra la relación entre estas colecciones ya explicadas:

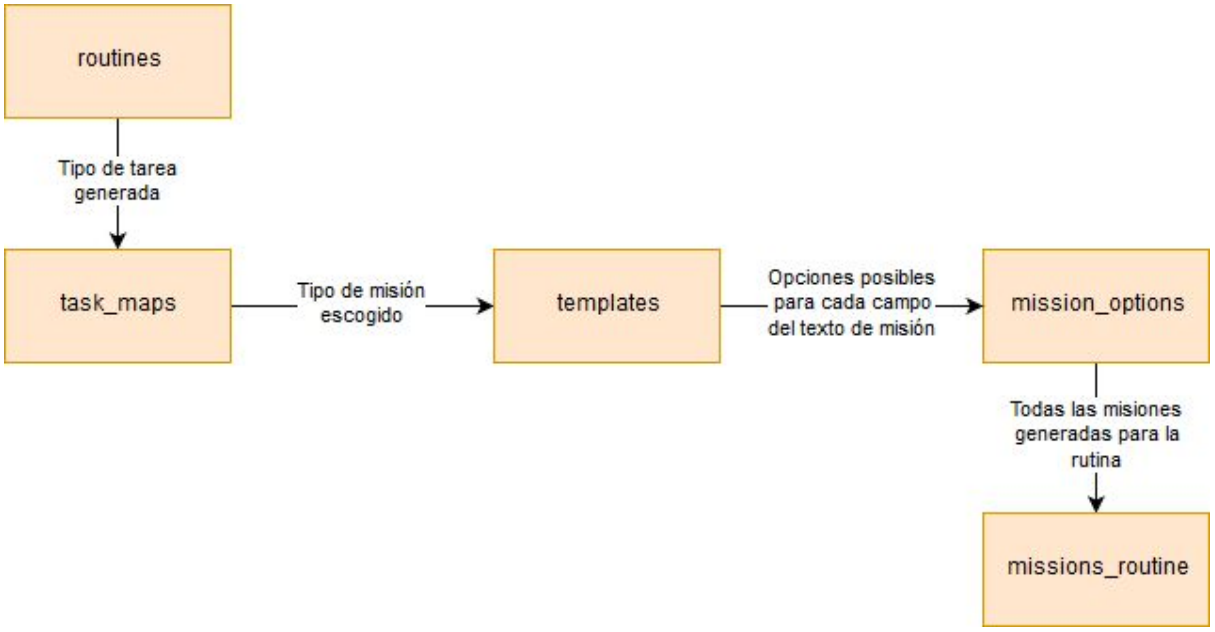

Figura 13: Diagrama de la base de datos del microservicio de rutinas.

Frontend

Tecnologías empleadas

Para la parte frontal del sistema se ha usado el framework **IONIC** que facilita al máximo la creación de estos de forma que, además, tenga un estilo con transiciones, colores definidos, estructuras propias por elementos, se genere en base al tamaño de la pantalla, etc.

IONIC define sus propias etiquetas para los elementos HTML, normalmente siendo la etiqueta normal pero añadiendo ion al principio como se puede apreciar en la imagen de abajo que muestra una pequeña parte del código de una de las páginas del sistema.

```
<ion-item>
    <ion-label floating color="primary" stacked>Search App User</ion-label>
    <ion-input type="text" (input)="searchForAppUser($event, searchKey)"></ion-input>
</ion-item>
```

Figura 14: Ejemplo de código IONIC.

La razón de usar dicho framework reside en la falta de conocimiento y experiencia en la creación de frontales, por lo que primó a sencillez, rapidez y elegancia a la hora de escoger cómo implementar el frontal de este sistema, teniendo en cuenta que para desarrollar el frontal con IONIC únicamente hace falta definir las etiquetas HTML dado que el estilo es aceptable para usarlo se tomó la decisión definitiva de uar IONIC para el desarrollo. Un punto que también fue muy importante a la hora de escogerlo como framework de desarrollo fue que, a futuro, se sabía que se iba a usar este para la implementación de la aplicación móvil dado que no se tenía experiencia en desarrollo con código nativo por lo que, al usarlo también aquí, se tendrá experiencias de usuario similares y, además, evita aprender dos frameworks diferentes para la misma tarea.

Aunque se haya usado IONIC para la creación del frontal, se sigue requiriendo de código para la comunicación de este con el *backend* y modificar la vista de información al usuario en base a la que se recibe del sistema, por lo que también se tuvo que programar usando el lenguaje Angular.js que no solo permite definir variables que se carguen en el frontal (creando así las páginas del frontal de manera dinámica) si no que también se pueden definir condiciones para que se carguen uno u otro elemento en función de la necesidad, recorrer listas de datos y mostrarlos en el frontal, definir la comunicación con el sistema usando llamadas HTTP, etc.

Por último y aunque los cambios realizados hayan sido mínimos, para la creación de páginas de estilo se ha usado SASS, un lenguaje que permite definir páginas de estilo CSS pero usando ideas de más alto nivel como declaración de variables.

En resumen, para la creación de la parte frontal del sistema se ha usado el framework IONIC para facilitar al máximo la tarea y el lenguaje Angular para añadir dinamismo y comunicación a la estructura que genera IONIC.

Arquitectura

La aplicación **Frontend** del sistema se estructura usando dos elementos importantes, las páginas y los proveedores. Las páginas serán las que muestren al usuario la información deseada y todas las funcionalidades que tiene disponibles para interactuar con esta mientras que, los proveedores, son los que ofrecen a las páginas una interfaz de comunicación con el **Backend** de forma que es capaz de utilizar los servicios que este tiene desplegados mediante llamadas HTTP que realiza el proveedor. El flujo de comunicación desde que el usuario realiza una acción hasta que obtiene la respuesta se puede apreciar en la Figura 15.

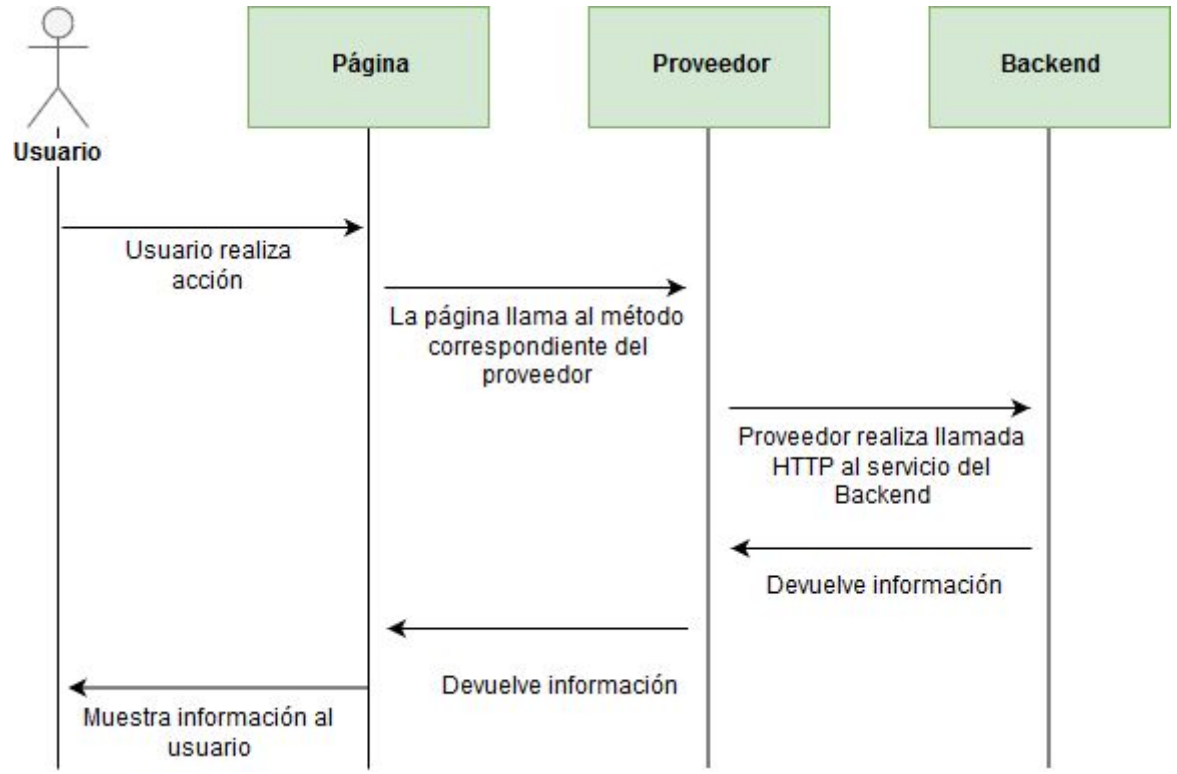

Figura 16: flujo de comunicación entre elementos de Frontend y Backend.

Páginas

Internamente, las páginas siguen el patrón MVC que usa IONIC, de forma que la página HTML es la vista, el controlador será el fichero ts (TypeScript) mientras que el modelo será el definido como modelo de datos en las comunicaciones con Backend (JSON de retorno de los servicios).

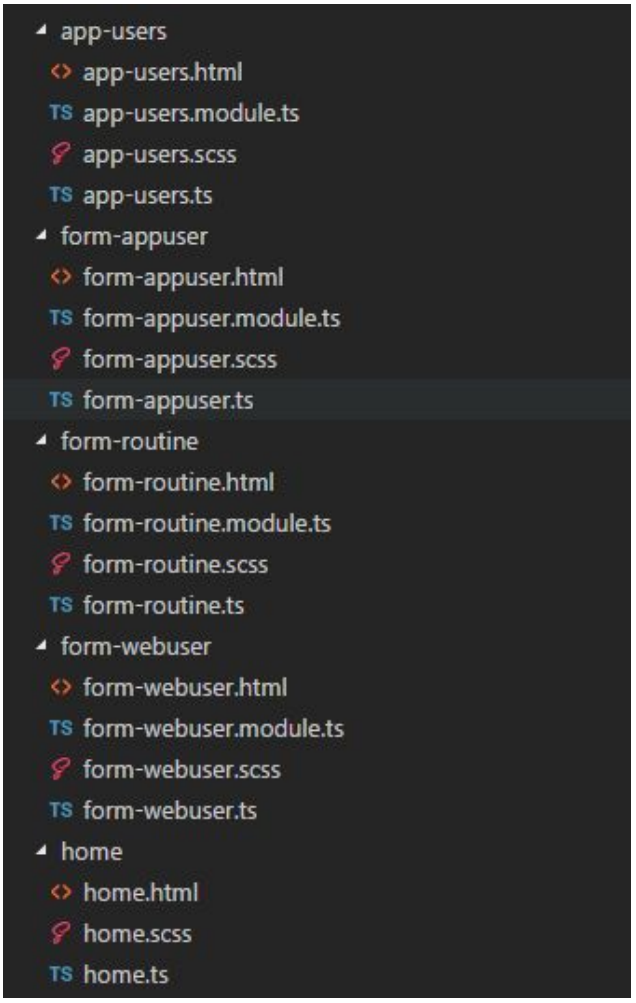

Figura 17: Ejemplo de estructura MVC en algunas páginas.

La vista podrá acceder a los elementos que tiene definidos el controlador como variables de forma que el contenido generado y renderizado por Angular se crea de forma dinámica usando información definida en ejecución, como puede ser información obtenida de un servicio, información del usuario con el que se hizo login, etc. La información obtenida a partir de los servicios se almacena con una estructura definida previamente con los modelos, cada modelo está definido como una interfaz donde se especifica qué campos tiene almacenados y de qué tipo son como se puede apreciar en la imágen del modelo de *IndoorSettingsModel.*

```
interface IndoorSettingsModel{
    url : string ;
    manufacturer : string;
    username : string;
    password : string;
}
```

Figura 18: Ejemplo de modelo en el frontend.

Proveedores

Los proveedores implementados en la aplicación, actúan como una interfaz de comunicación con los servicios definidos en el Backend, tienen implementadas funciones para la comunicación con este y procesamiento de los datos retornados por estos para que, las páginas, las invoquen cuando quieran obtener datos del recurso del proveedor.

```
searchWebUsers(data){
  return this.http.get(this.BASE_URL_WEB_USER_API);
}


addWebUser(data){
  return new Promise((resolve, reject) => {
    this.http.post(this.BASE_URL_WEB_USER_API, JSON.stringify(data), {headers:{'Content-Type':'application/json'}})
      .subscribe(res => {
        resolve(res);
      }, (err) => {
        reject(err);
      });
  });
}

modifyWebUser(id, data){
  return new Promise((resolve, reject) => {
    this.http.put(this.BASE_URL_WEB_USER_API + '/' + id, JSON.stringify(data), {headers:{'Content-Type':'application/json'}})
      .subscribe(res => {
        resolve(res);
      }, (err) => {
        reject(err);
      });
  });
}
```

Figura 19: Ejemplo funciones del proveedor de usuarios de aplicación web.

Estructura de páginas

Las páginas en el frontend tienen una estructura de niveles, primero se tiene una página de login obligatoria para poder acceder al sistema que consiste en un formulario sencillo para introducir las credenciales junto con la opción de registrarse como nuevo usuario y la opción de resetear la contraseña.

Figura 20: Página de login para usuarios aplicación web.

Una vez se identifica y puede entrar en la aplicación, la página se divide en cuatro apartados principales, **Home, Web Users, App Users** y **Routines** donde en cada apartado lo primero que se muestra es el listado de elementos de dicho recurso junto con una opción de búsqueda para filtrar y un botón para añadir uno nuevo si se permite.

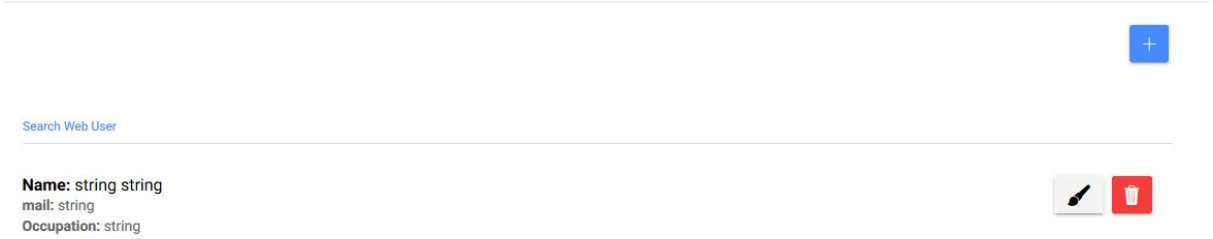

Figura 21: Página de listado de usuarios web.

A partir de aquí se puede realizar varias acciones por lo que el flujo deja de ser dirigido:

- Se puede ver la información completa de uno de los elementos concretos al realizar click sobre él.
- Se puede editar la información de uno de los elementos con un formulario pulsando en el botón con el icono de un pincel.
- Se puede borrar un elemento pulsando el botón con el icono de una papelera.

Las últimas dos opciones también se tienen disponibles directamente en la página de mostrar la información completa de un elemento.

Aplicación móvil

Tecnologías empleadas

En la implementación de la aplicación móvil se han usado las mismas tecnologías que para la parte frontal del sistema, en este caso las razones para elegir este framework aunque son las mismas existe una principal y es que no se tiene conocimiento previo en el desarrollo de aplicaciones móviles usando código nativo ya sea Android o Ios. Aprender a implementar usando código nativo no es una tarea sencilla o que se pueda aprender en poco tiempo, además teniendo en cuenta que se debería implementar tanto para Android como para Ios al desconocer el tipo de móvil del que disponen los pacientes.

Para permitir entonces la creación de la aplicación móvil, para ambos sistemas operativos usando el mismo código, se tomó la decisión de usar IONIC dado que dispone de esta funcionalidad.

Arquitectura

La aplicación móvil de igual manera que la aplicación del frontal del sistema está generada usando la arquitectura MVC de Ionic, es decir, tiene modelos definidos con interfaces, la vista hace referencia a la página HTML dinámica que se genera en función de la información que se carga por el controlador de esta que es el fichero de extensión ts (typescript) que es el encargado de comunicarse con los *providers*, obtener los datos necesarios y transformarlos de la forma requerida para mostrarlos al usuario.

De igual manera que en la parte del frontal, la primera página a la que se accede es una página de login donde el usuario de la aplicación móvil debe introducir sus credenciales para acceder pero, al contrario que en el frontal, aquí se dispone de una página de registro que permite a un usuario nuevo crear una cuenta nueva con su información asociada.

Una vez acceda con las credenciales se le presenta la página *home* dándole la bienvenida a la aplicación y da acceso a un menú usando el botón de la parte superior izquierda, botón que se encuentra disponible en todas las demás páginas del sistema que sean de acceso raíz.

> Para aclarar esto, una página raíz es una página central de la aplicación, se verá después que existen páginas a las que se accede a través de una página raíz como un formulario de creación y, dicha página, no dispone del menú si no de un botón para volver atrás a la página raíz que sí dispone de esta funcionalidad.

Las páginas raíz del sistema son las expuestas con el menú mencionado, se explican una a una a continuación:

- ***Home*** → la página *Home* es la página de inicio del sistema, únicamente da la bienvenida al usuario.

- ***Profile*** → la página de *Profile* permite al usuario que se ha autenticado con éxito en la aplicación mostrar su información personal relacionada con este perfil, concretamente permite examinar la siguiente información:
    - Nivel del usuario actualmente en el sistema.
    - Rol que ha escogido el usuario al crearse la cuenta (Ej. Mago, Guerrero, etc).
    - Plantilla que ha escogido al crearse la cuenta (Ej. Pirata, Astronauta, etc).
    - Información propia de contacto con el usuario:

        - Correo electrónico con el que se ha registrado
        - Número de teléfono
        - Alias de Telegram para futuras comunicaciones

    La página permite además modificar la información relevante **únicamente** a contacto con el usuario, es decir, el correo electrónico, el número de teléfono y el alias de telegram.

- ***Missions*** → la página de misiones permite al usuario analizar las misiones / tareas que tiene que realizar el día de hoy, la hora a la que tiene que realizarlas, de que tipo son, si tiene que realizarla ahora, está cumplida o no, etc. Dado que la información de las misiones se generará cuando los usuarios administrativos creen una rutina para el usuario pero este puede estar en la aplicación en cualquier momento, se ha usado una funcionalidad de IONIC que permite actualizar manualmente las misiones, dicha funcionalidad es el denominado **Refresher**.

A parte de listar las misiones que tiene que realizar el día de hoy, también permite entrar a ver en detalle una misión concreta, de esta forma se mostrará al usuario toda la información relevante a la misión además de un botón para iniciar el proceso de completarla siempre y cuando la hora a la que tiene que realizarla coincida con la hora actual. A continuación se listará la información que permite visualizar la página de detalle de misión:

- Tipo de la misión (se recuerda hace referencia a Rescate, Infiltración, etc).
- Estado de la misión (completada, incompleta)
- Hora de inicio de la misión.
- Hora de finalización de la misión.
- Texto de la misión explicando el objetivo de esta y las recompensas que se pueden obtener.

Una vez el usuario selecciona el empezar el proceso de cumplir la misión se muestra una página nueva, la que presenta al usuario con la imagen del mapa del piso en el que se encuentra actualmente junto con su posición usando un icono de posicionamiento y, para

guiar al usuario se mostrará la distancia que le queda por recorrer junto con una flecha indicando en la dirección que tiene que moverse.

- ***Friends*** → la página de *Friends* es la que permite acceder al listado de amigos que tiene el usuario que se ha autenticado en la aplicación, estos amigos son otros usuarios de la aplicación móvil con los que tiene relación y ha decidido añadirlos a su lista. La información mostrada en el listado para cada amigo es la siguiente:

    - Alias del usuario.
    - Nivel que tiene actualmente.
    - Rol que ha escogido ser.

    En esta página se dan dos opciones adicionales para el flujo, la primera es eliminar un amigo de la lista usando el botón de la parte derecha que está alineado con cada amigo y, la segunda opción, es añadir un nuevo amigo a la lista pulsando el botón de **'+'** que está situado arriba a la derecha, esto hará que se muestre un listado con todos los usuarios móviles de la aplicación excluyendo evidentemente al propio usuario y a los que ya se encuentran en su lista de amigos y, al seleccionar cualquiera de los listados se añadirá automáticamente a la lista.

Por último se muestra un diagrama con todo el flujo posible que se puede tener por parte del usuario en la aplicación móvil.

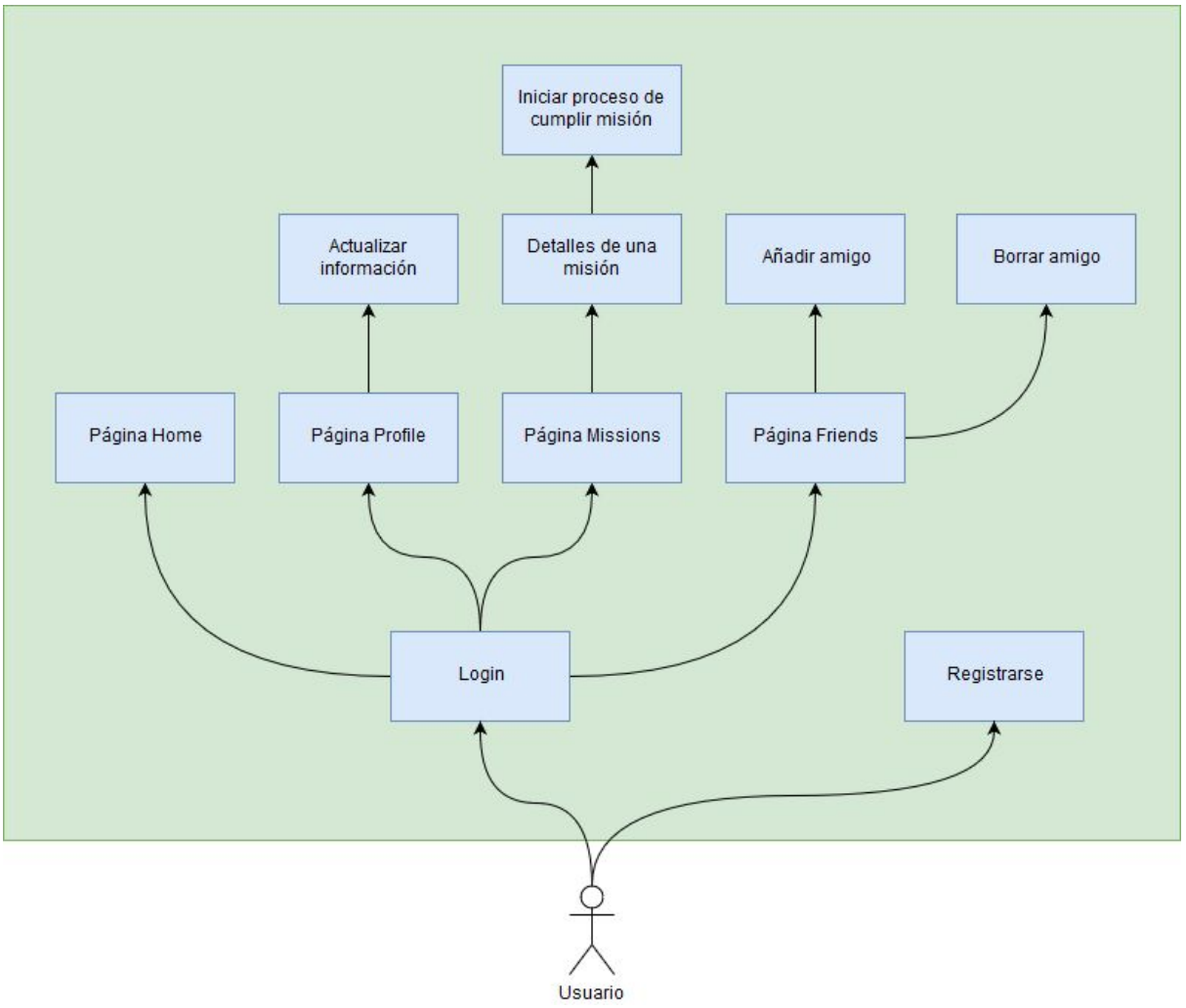

Figura 22: Diagrama de flujo de la aplicación móvil.

Bot

Para la implementación del bot se ha usado el lenguaje de programación Node.js dado que es uno de los lenguajes que acepta Telegram para la creación de estos. Se ha usado frente a otros por su facilidad para crear un servidor web con **express** lo cual era uno de los requisitos para el bot, rapidez y elegancia del código generado y facilidad de aprendizaje para programar.

La duda existía entre usar este lenguaje y Python debido a que cumple en mayor o menor medida las razones por las que se escogió Node y además se tenía conocimiento previo en el desarrollo usando este lenguaje, sin embargo, analizando benchmarks que comparan el rendimiento de una serie de programas en ambos lenguajes se puede apreciar que los que están implementados usando el lenguaje de Node ejecutan bastante más rápido que los de Python, por lo que al final se decantó por usar **Node.js**.

Como ya se ha nombrado, para la creación del servidor web se ha usado el *framework* **express** que está basado en *sinatra*, otro framework para aplicaciones web. Aunque en el sistema se haya usado únicamente para el despliegue de un servidor web, este framework permite bastantes más cosas, entre las que esta gestión de sesiones, tratamiento de cookies, tratamiento de llamadas, etc.
Para usar el framework basta con tener instalado *npm* y ejecutar el comando de instalar express, además, para mostrar lo sencillo que es desplegar un servidor web usando este framework, se muestra abajo todo el código necesario:

```
const app = express();
app.use(bodyParser.json());

app.get('/', function (req, res) {
  res.json({ version: packageInfo.version });
});

var server = app.listen("8084", "0.0.0.0", () => {
  const host = server.address().address;
  const port = server.address().port;
  console.log('Web server started at http://%s:%s', host, port);
});
```

Figura 23: despliegue del servidor web con express.

En cuanto a la comunicación con el bot en Telegram se ha usado el módulo **node-telegram-bot-api** que se encuentra disponible en un repositorio del mismo nombre en Github. Este módulo permite interactuar con la API oficial de Telegram y, de esta forma, interaccionar con el bot desplegado en esta aplicación; para desplegarse únicamente necesita un token de bot que se genera cuando se crea uno usando el servicio propio de Telegram de creación de bots **BotFather**.

El módulo node-telegram-bot-api es uno de los más completos y útil ahora mismo para bots, ofrece una gran variedad de funciones de interacción, además de tener una amplia y completa documentación así como tutoriales donde se puede comprobar en código la forma de implementar cada elemento.

Partes importantes del sistema

Servidor de posicionamiento interno

El sistema dispone de un microservicio que permite el posicionamiento de los pacientes en el hospital usando la aplicación móvil implementada. Este microservicio se usa cuando el paciente comienza el proceso de completar una misión, la cual tiene un identificador de zona / nombre de habitación) donde tiene que desplazarse, por lo que se despliega en la aplicación móvil un mapa del piso donde está actualmente el paciente y se inicia el proceso de guiarle hasta su destino.

Arquitectura necesaria en el hospital

Para las pruebas realizadas al sistema de posicionamiento interno del sistema se ha usado la tecnología WIFI de Cisco que, mediante la arquitectura de Connected Mobile Experiences (CMX) ofrece una serie de servicios con los que un sistema es capaz de obtener una gran cantidad de información de la red WIFI donde se tiene desplegado, por ejemplo:

- Información detallada sobre los usuarios activos en la red
- Información de los mapas del edificio donde se ha desplegado la red WIFI
- Información de presencia en el edificio
- Variada información de analíticas sobre los elementos de la red

Todos estos servicios se tienen como *endpoints* en una API REST que tiene el CMX que se puede apreciar en la imagen de abajo:

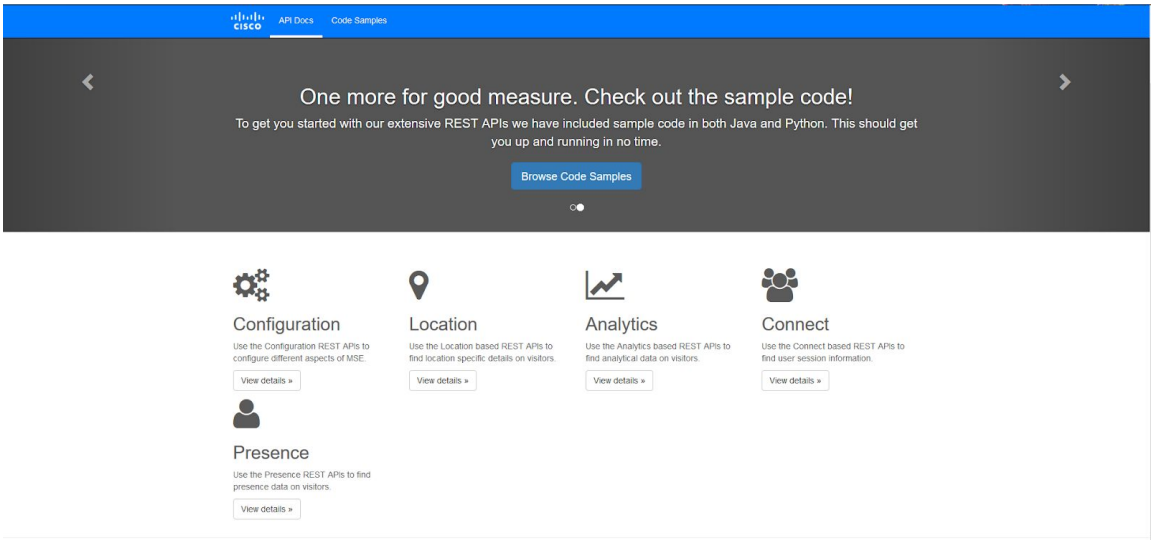

Figura 24 : API REST del CMX.

Para que este elemento del sistema funcione correctamente se necesitan una serie de requisitos de configuración que deben cumplirse:

- **Servicios de posicionamiento desplegados y configurados:** el requisito principal para que se pueda usar el sistema de posicionamiento es que se tenga desplegado los servicios necesarios para que sea posible. De nuevo, usando el caso de pruebas que es con tecnología Cisco, sería necesario desplegar en la red del hospital el servicio CMX y el Prime Infrastructure, que es el servicio que permite la carga de mapas y la configuración de estos para, posteriormente, posicionar a los usuarios mediante el CMX.

- **Estructura del edificio detallada:** una vez se tienen desplegados los servicios del fabricante escogido, es necesario que se defina de forma rigurosa la estructura del hospital para aumentar la precisión y utilidad del posicionamiento. Usando el ejemplo de Cisco, este tiene definición para tres tipos de estructura:
    - *Campus:* el campus, en el caso de este sistema, definiría el hospital entero.
    - *Edificio:* el edificio define una **estructura concreta** del campus definido, por ejemplo el hospital puede estar dividido en varios edificios separados.
    - *Piso:* hace referencia a un piso concreto de un edificio ya definido.

  De forma que queden definidos todos los pisos de todos los edificios del campus como tuplas *(Campus, Edificio, Piso)*.

- **Mapas cargados:** cuando se tiene definida toda la estructura del hospital es necesario que se cargue, por cada tupla antes definida, un mapa de su distribución interna para permitir situar a los usuarios en el mismo indicando su tamaño real. La imagen debe estar en formato **png** y debe tener como mínimo un tamaño de **512 x 512 píxeles.**

- **Zonas del edificio definidas:** cuando ya se ha completado todos los requisitos anteriores, como último paso es necesario definir con Prime Infrastructure las zonas/habitaciones de cada piso, de esta forma se podrá mapear las tareas definidas en las rutinas del paciente a una habitación concreta y se le podrá guiar a dicha habitación.

- **Despliegue de APs:** el último requisito necesario para que el sistema de posicionamiento funcione de forma correcta y precisa es que el edificio/hospital tenga un despliegue de APs tal que la potencia del WIFI sea óptima en todas las zonas del mismo, además, se recomienda el uso de APs que permitan el uso de **Hiper Localización** ya que aumenta la precisión del posicionamiento a 1 - 3 metros de diferencia, mientras que los estándar pueden llegar a tener una precisión de 20 metros.

Aunque en las pruebas se haya usado la tecnología de CISCO, existen tecnologías análogas por otros fabricantes como puede ser HP, HUAWEI, etc de forma que habría que cambiar la forma de conectarse a dicha tecnología y los modelos JSON que se reciben del servicio.

Funcionamiento en el sistema

El microservicio de localización del sistema se conecta con el servicio desplegado de localización del fabricante del Wifi del edificio donde se va a posicionar a los diferentes usuarios, tal y como ya se ha explicado, en las pruebas se ha trabajado con el fabricante Cisco el cual ofrece el servicio del CMX con diversos *endpoints* para usar en el posicionamiento interno de los usuarios.

Para la implementación del microservicio se ha hecho uso de cinco *endpoints*:

- **Información de estructura:** el primer *endpoint* que usa el microservicio es el que permite obtener la información de la estructura sobre la que se va a realizar el posicionamiento, en el caso de Cisco, la estructura como ya se ha explicado viene dada por *campus, edificio* y *piso*. Al usar este endpoint, el microservicio obtiene toda la información del hospital a nivel de arquitectura como edificio por lo que permite que, posteriormente, se permita a los usuarios web seleccionar una tupla concreta para asignar a una misión y al usuario de aplicación, ser guiado por esta estructura.

- **Información de mapa:** otro *endpoint* con el que es necesario que se comunique es el que permite obtener la información del mapa asociado a una tupla concreta de

campus, edificio, piso. La información que reporta este servicio y que, posteriormente, se usa por los diferentes elementos del sistema es la siguiente:

    - Número de *piso* al que hace referencia el mapa en el *edificio* y *campus* de la tupla.
    - Información sobre las dimensiones del piso:
        - Anchura
        - Altura
        - Unidad de medida
    - Información sobre la imagen del mapa:
        - Anchura
        - Altura
    - Listado de zonas / habitaciones que tiene el piso.

- **Obtener la imagen de un mapa:** para el posicionamiento del usuario es requisito mínimo el permitir al sistema obtener la imagen del mapa donde se le va a posicionar y por el que se le guiará una vez sea necesario. Para facilitar el trabajar con la imagen el microservicio una vez se comunica con el *endpoint* del fabricante para obtener la imagen, codifica su contenido usando **Base64** para permitir el envío por HTTP sin problemas de codificación y para facilitar el mostrar dicha imagen por la aplicación móvil una vez sea necesario.

- **Obtener la información de un usuario:** el último *endpoint* que usa el sistema es el de obtener la información relevante a un usuario concreto por su IP otorgada por la red Wifi. Este servicio permite obtener una serie de información que será después usada para situar al usuario en un punto concreto del mapa y permitir la visualización de esto por la aplicación móvil:

    - Tupla de *campus, edificio, piso* donde se encuentra localizado el usuario en este momento.
    - Coordenada exacta de la posición de usuario (x,y,z,unidad de medida).
    - Coordenada geográfica de la posición del usuario (latitud, longitud, unidad de medida).

El flujo de comunicación de las aplicaciones que tienen interacción directa con el usuario (aplicación web y aplicación móvil) con el microservicio de localización viene relejado en las diagramas de abajo, como se puede apreciar cada aplicación hace uso de una serie de servicios que este ofrece.

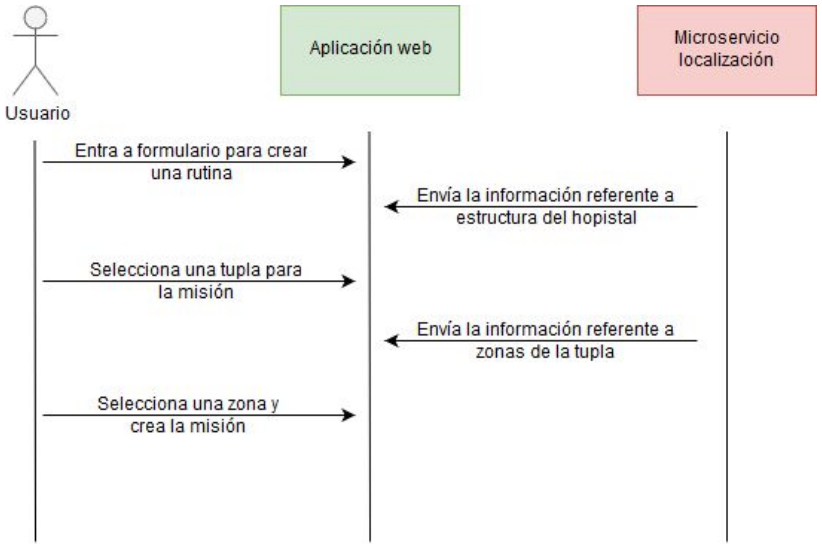

Figura 25: Diagrama de flujo de aplicación web con microservicio.

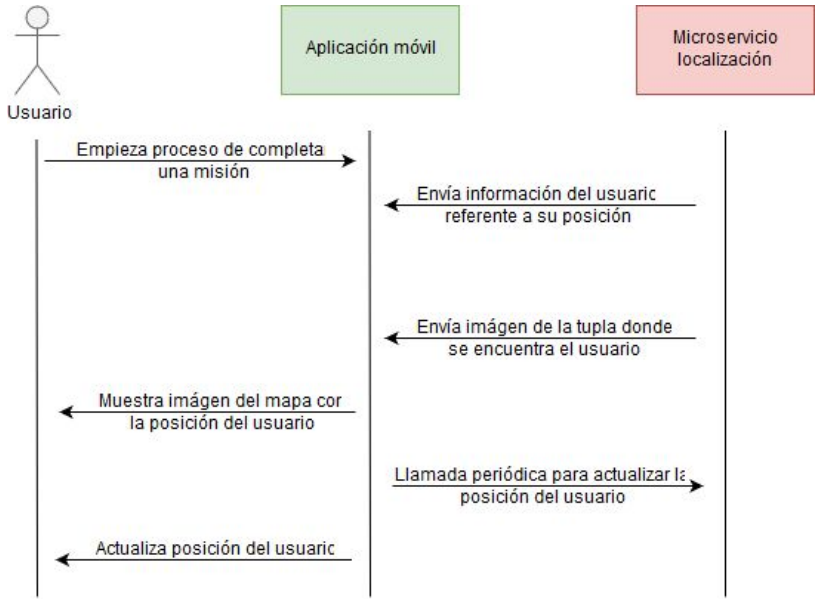

Figura 26: Diagrama de flujo de aplicación móvil con microservicio.

Generador de misiones

Estudio de diferentes tipos de misiones

Antes de pasar a la implementación del generador de misiones del sistema, se comenzó planteando los diferentes tipos de misiones que debían existir en función de las diferentes tareas que podían tener los pacientes sacadas de la información obtenida en la visita al hospital. Las diferentes tareas han quedado con la siguiente estructura:

Tipo	Subtipo	Alcance
Colegio	X	El tiempo destinado por el hospital a que los niños vayan al colegio.
Evento	Visita Evento para ver Evento para participar	Engloba toda aquella tarea generada por terceros como pueden ser visitas de voluntarios, visitas especiales con temática, eventos como obras de teatro o charlas donde solo van a ver o eventos para participar como juegos.
Prueba médica	En habitación En sala	Cualquier prueba médica realizada a los niños. Se divide en pruebas realizadas en la habitación o en una sala especializada (falta estudiar si en sala sería también que vaya el médico o vaya el paciente).

Las misiones generadas a partir de las tareas arriba definidas pueden ser de varios tipos:

- **Rescate**: esta misión necesitará la ayuda de un NPC que ponga un cierto objeto en una sala de forma que el objetivo del usuario es llegar a esa cierta sala donde tiene la tarea y, anteriormente, el NPC dejó el objeto y obtenerlo.
    - Posible evolución: el usuario que recata el objeto se lo puede quedar como recompensa (promociones especiales).
    - Posible evolución II: esta tarea puede evolucionar a una segunda que sea que el propio usuario esconda el objeto en otra sala para generar una nueva misión en un segundo usuario diferente.

- **Infiltración/Espionaje:** esta misión no necesita la ayuda de ningún NPC asociado y tiene como objetivo que el usuario acceda a una sala un cierto período de tiempo.
    - Posible evolución: se puede asociar un NPC de forma que el objetivo de la misión sea obtener cierta información y contestar en la aplicación (bastante relacionado con el colegio).

- **Captura (de zona o de objeto):** esta misión no necesita de ningún NPC y el objetivo es que el usuario esté en una cierta sala un tiempo concreto para “conquistarla”. necesita de un servicio de posicionamiento fiable para saber que el usuario está en la zona.

- **Entrega:** el objetivo de esta misión es entregar un cierto objeto a una persona u zona. Necesitará de un NPC asociado siempre y cuando el objeto se entregue a una persona concreta mientras que, si se entrega a una zona, no haría falta de nada. Haría falta un comando en el bot para saber que se entregó a un NPC el objeto (botón / contraseña) o un indicador de zonas si es en un sitio.

- **Recolección:** el objetivo de la misión consiste en recolectar una serie acciones, por ejemplo en el colegio obtener todas las respuestas bien.
    - Posible evolución: pedir a un NPC que escoja una serie de objetos y grabe en cada uno de ellos una letra que formará una palabra decidida de antemano por el sistema. El NPC deberá indicar cuáles son los objetos elegidos y entónces el usuario deberá encontrarlos e introducir la palabra en la aplicación.

- **(A partir de ll evolución de rescate) Ocultación:** No necesita un NPC, el objetivo del usuario es coger un objeto que tenga y esconderlo en una zona. Tras esconderlo deberá reportar en la aplicación qué objeto era.

El mensaje de la misión estará formado por 4 partes:

Parte	Opciones
Saludo	Querido {user.alias} {user.alias} necesitamos tu ayuda Tienes una nueva misión {user.alias} {user.alias} hay algo nuevo
Cuerpo	*Subdividir*
Despedida	Eso es todo. Ayuda por favor. Espero que no tengas problemas. Estaremos en contacto. Gracias por todo.
Recompensas	Experiencia [formada por el 5% necesario para subir de nivel más un valor variable entre el 0% y 3% extra] Dinero [número aleatorio con máximo nivel*13+6]

Implementación del generador

Como se ha comentado anteriormente, el microservicio del recurso rutinas no solo tendrá las operaciones CRUD sobre el recurso, si no que también tendrá el generador de misiones implementado, un componente que se encargará de producir textos pseudoaleatorios para las misiones de los usuarios de aplicación móvil originados a partir de cada tarea que tenga la rutina. El funcionamiento será el siguiente:

1. El sistema obtendrá una tarea de la rutina que se le ha planteado al usuario y enviará el tipo de la misma al generador. Se recuerda que los diferentes tipos de tareas son:

    - Colegio
    - Evento
        - Visita
        - Evento únicamente para ver
        - Evento para participar
    - Prueba médica
        - En la habitación del usuario
        - En sala de médico

2. El generador usará el tipo de tarea recibida para obtener las posibles misiones en las que puede convertirse, esta relación estará definida en base de datos de forma

que se facilite el añadir nuevas tareas. Por ejemplo la tarea *Colegio* podría convertirse en los siguientes tipos de misiones:

- Infiltración/Espionaje
- Captura (zona u objeto)
- Recolección

3. Una vez seleccionado el tipo de misión que se va a generar, se accede a base de datos para obtener la plantilla de la misma junto con las posibilidades para cada uno de los apartados. Siguiendo el ejemplo de la misión para *Colegio* supongamos que se seleccionó el tipo *Infiltración*:

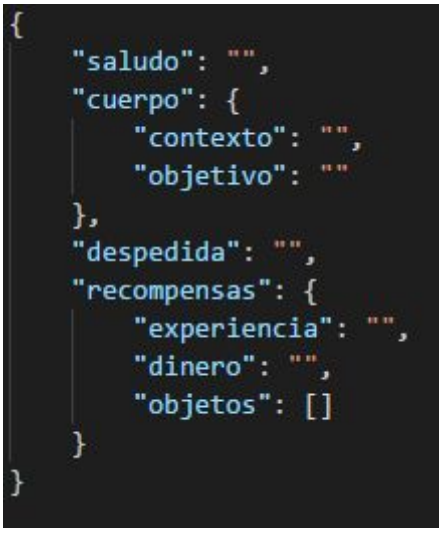

Figura 27: Plantilla de misión Infiltración con sus opciones.

Como se puede apreciar en la Figura, el texto de la misión se dividirá en un saludo, un cuerpo de la misión que se subdivide en el problema a resolver junto con el objetivo de la misión y para finalizar una despedida junto con las recompensas, que, en este caso, se han definido como experiencia, dinero y objetos.

Las posibilidades para rellenar esta plantilla se encuentran también en base de datos y consiste en una lista de posibles textos para cada apartado, de forma que el número de posibilidades para los textos de misión se convierte en la permutación de todos estos elementos. El funcionamiento será que el generador irá obteniendo números pseudoaleatorios para seleccionar en cada apartado cuál es la opción que escoge en cada uno, además, se tendrá en cuenta la plantilla escogida por el usuario de forma que se pueden generar textos de estilo **pirata, astronauta** o **vaquero** por ejemplo**.**

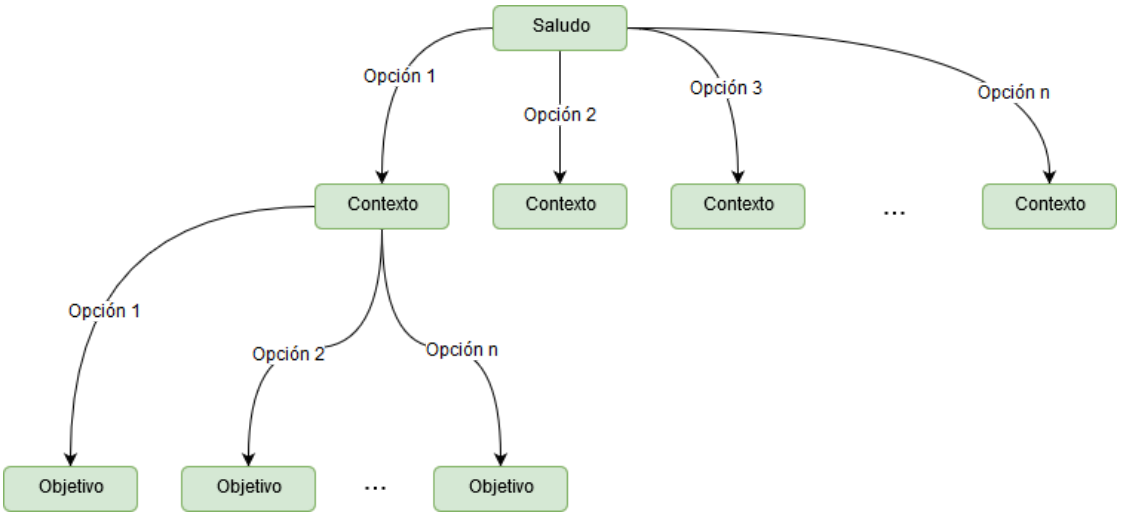

Figura 28: Árbol de decisión para el texto de las misiones.

4. Las recompensas se calcularán de diferentes maneras, para la experiencia se basará en un porcentaje de la experiencia total necesaria para subir de nivel, el dinero será un dinero base normal más una parte variable que se generará como un número pseudoaleatorio contenido en un rango de valores definidos por nivel de usuario. Los objetos por último se dejará como algo especial de eventos al tratarse de objetos físicos en el mundo real y se permitirá añadir a la hora de definir la tarea por el responsable.

5. Por último, una vez el generador haya seleccionado opción para cada apartado, unirá todo y lo devolverá como el texto final de la misión.

Arquitectura de despliegue

Docker

¿Qué es Docker?

Docker consiste en una herramienta para construir y desplegar Software de una forma encapsulada con todas las dependencias necesarias para que funcione de forma correcta. Esta idea de encapsulamiento se realiza usando la unidad de despliegue que usa Docker llamada contenedor, un contenedor es una pieza de software que incluye todo lo necesario para ejecutarse por sí solo (código, herramientas y librerías del sistema, ajustes, etc) que está disponible para aplicaciones de Linux y Windows.

Beneficios de Docker

El uso de Docker puede traer varias ventajas al despliegue de Software:

- **Estandarización y productividad** → mediante el uso de contenedores Docker se asegura la consistencia entre diferentes ciclos de desarrollo y de publicación. La estandarización se tiene dado que Docker permite tener entornos de desarrollo, construcción, pruebas y producción iguales para cada miembro del equipo de forma que todo se realice con las mismas variables y reduzca al máximo los problemas de unificación de código.
  Además, Docker tiene un entorno exclusivamente montado para subir imagenes y cambios en estas que las demás personas se pueden descargar, se trata de un Git únicamente dedicado a imágenes de Docker. De esta forma se puede mantener un control de versiones no solo en el código si no también en el entorno donde el código se va a ejecutar.

- **Eficiencia** → Docker permite construir una imagen de contenedor y usarla para cada paso del proceso de despliegue. Un beneficio de esto es que permite separar pasos que no tengan dependencias entre sí y ejecutarlos en paralelo.

- **Compatibilidad y mantenimiento** → Las imágenes ejecutan de la misma manera y con el mismo comportamiento da igual el sistema donde estén, sea Windows, Linux, Mac, da igual la versión del mismo. Esto lleva a que se necesite menos tiempo para montar el entorno de desarrollo dado que se monta una vez y ya está y, además, se tiene más portabilidad en el entorno.

- **Agiliza el desarrollo** → dado que Docker no tiene que ejecutar un sistema operativo, únicamente lanzará contenedores, agiliza en gran medida el despliegue de los proyectos.

- **Aislamiento** → mediante el uso de Docker se asegura que las aplicaciones y los recursos de esta están aislados dado que Docker se asegura que cada contenedor tiene sus recursos aislados de los demás contenedores de forma que no puedan acceder. Esto permite que se tenga un proyecto con varios contenedores donde cada uno tiene su configuración y recursos propios, además, que facilita la tarea de eliminar una aplicación, como está totalmente aislada y ningún otro contenedor usa sus recursos, para eliminarla basta con eliminar el contenedor.

- **Seguridad** → dado que Docker se asegura que las aplicaciones ejecutando en contenedores estén completamente aisladas, eso da control sobre el tráfico de red y el mantenimiento del mismo.

Docker en el sistema

En el sistema creado, Docker se utiliza para el despliegue de cada microservicio definido así como para la parte frontal del sistema y el bot que se usa para la comunicación.

La creación de las imágenes de Docker se ha realizado usando ficheros Dockerfile donde se define las instrucciones que debe seguir para generar dicha imagen. Los ficheros de los microservicios son prácticamente iguales:

```
FROM maven:3.5-jdk-8-alpine
WORKDIR /app
COPY /. /app
RUN mvn install

FROM openjdk:8-jre-alpine
WORKDIR /app
COPY --from=0 /app/target/web-users-microservice-0.0.1-SNAPSHOT.jar /app
CMD ["java -jar web-users-microservice-0.0.1-SNAPSHOT.jar"]
```

Figura 29: Dockerfile del microservicio web.

Como se puede apreciar en la imagen, se usa el concepto de *multi-staging* de Docker en el que se parten de varias imágenes en cada etapa de la construcción, en la primera etapa se usa una imagen de Maven para generar el ejecutable mientras que en la segunda se copia el fichero *jar* generado y se ejecuta con java en una máquina de este tipo. Esto evita tener ficheros Dockerfile con un gran número de instrucciones e instalaciones complicadas.

El Dockerfile del frontend y del bot son más sencillos, ambos parten de una imagen tipo node dado que ambos usan express como servidor para desplegarse, en ambos casos además, se copia todo lo contenido en esa carpeta a la ruta */app* y, después, se instalan las dependencias y se ejecuta el servidor con la instrucción *node*.

Por último están las imágenes de las bases de datos de cada contenedor que tiene desplegado un microservicio, estas imágenes están creadas a partir de un ubuntu versión 16.04 al que se le instala mongo. La única diferencia entre las imágenes es que en la imágen de base de datos para el microservicio de usuarios web se tiene que insertar un usuario administrador para realizar el login y poder crear a los demás usuarios. Todas las imágenes tienen como última instrucción con CMD la instanciación de un servicio Mongo con un puerto concreto para recibir llamadas de los microservicios.

Persistencia de las imágenes: Dockerhub

Para poder acceder a las imágenes creadas en cualquier momento y facilitar esta tarea se ha creado un repositorio en el servicio de DockerHub y se han subido aquí. De esta forma no se tienen que tener creadas de forma local en el ordenador antes de realizar el despliegue con kubernetes, directamente se tiene preparado para que realice pull de estas imágenes y, así, se minimiza el número de pasos a realizar antes de desplegar el proyecto.

Kubernetes

¿Qué es Kubernetes?

Kubernetes es una plataforma creada por Google que permite la gestión de despliegues en contenedores Docker y servicios que facilita tanto la configuración de estos como la automatización de despliegues, replicación de estos, etc.

Kubernetes proporciona un entorno de gestión centrado en contenedores. Organiza la computación, las redes necesarias y configuración de esta, la infraestructura de almacenamiento para las cargas de trabajo de los usuarios.

Beneficios de Kubernetes

El uso de Kubernetes viene dado por la funcionalidad que ofrece a la hora de automatizar despliegues de aplicaciones complejos donde se necesita el tener varias máquinas independientes unas de otras pero que se puedan comunicar entre sí de una forma sencilla, además, también es importante el poder configurar estas máquinas cada una con su estado propio; Kubernetes permite todo lo anterior expuesto de una forma sencilla al permitir el uso de ficheros de extensión **YAML** para realizarlo.

Kubernetes en el sistema

En el sistema propuesto se han usado cinco estructuras básicas de Kubernetes:

- **Cluster:** el cluster en Kubernetes es donde se va a desplegar toda la arquitectura del sistema, está formado por un o un conjunto de nodos siendo un nodo cualquier dispositivo con capacidad de cómputo (ordenador, reloj inteligente, móvil, etc) de forma que es una malla de dispositivos que ofrecen su capacidad de cómputo para que un sistema sea capaz de funcionar. En el caso del sistema se ha usado Minikube, un cluster ofrecido por Kubernetes que se despliega en local usando una máquina virtual y que dispone de un único nodo, el propio host donde se despliega.

- **Pod:** un pod se puede definir como una estructura que junta uno o varios contenedores Docker de forma que comparten los mismos recursos y red local, por lo que se pueden comunicar de una forma muy sencilla como si estuvieran ejecutando en la misma máquina pero manteniendo independencia entre ellos. Además, los Pods son la unidad de replicación en Kubernetes, permite que, si se necesita más capacidad de carga por el nivel de usuarios, se puede crear nuevas réplicas de todo el contenido del Pod, lo que además permite balanceo de carga y resistencia a errores.

- **Deployment:** un Deployment es una estructura que actúa como una capa abstracta entre un conjunto de pods que se despliegan y el cluster donde lo hacen, el beneficio

de usar Deployment es que se puede definir el número de réplicas que se quieren de un pod y este los creará automáticamente al estar en el cluster, además, si ocurre algún fallo en algún pod, el deployment lo recreará de forma automática e invisible al usuario.

- **Service:** un service es especialmente útil cuando es necesario que los pod se comuniquen entre ellos o que un tercero realice peticiones a alguno, esto es porque los Pod son efímeros, se pueden destruir y cuando ocurra el deployment se encargará de reconstruirlos pero eso significa que, si queremos comunicarnos por IP puede ocurrir que no siempre tenga la misma al reconstruirse y es por esto que los Service son tan importantes, ya que permiten añadir una política de acceso al Pod mediante una etiqueta definida en la configuración; ell uso del Service además permite que, si se tiene varias réplicas de un Pod siempre se sepa como llegar a cualquiera de estos.

- **Ingress:** la última estructura usada es el Ingress, es la que permite el tráfico originado del exterior a los servicios desplegados en el sistema, así no solo se pueden comunicar los diferentes servicios desplegados si no que también se pueden tener sistemas externos que interactúan con el sistema.

  Existen varias formas de permitir el tráfico de entrada, NodePort, LoadBalancer o un Ingress, la razón de que se haya usado esta última es que es la que más potencia de configuración permite, teniendo la funcionalidad añadida de un DNS propio de forma que para comunicarse con los diferentes servicios del sistema no se usa direcciones IP si no que se usan nombres de dominio. Para permitir el uso de un Ingress es necesario desplegar un Ingress Controller y en el caso del sistema al hacer uso del cluster Minikube se ha usado el addon propio de este.

Para definir todo lo anterior se ha hecho uso de ficheros de tipo YAML que, como se puede apreciar en la imagen de abajo, la sintaxis es muy parecida a JSON y las palabras clave de Kubernetes hace que el fichero sea bastante sencillo de entender.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: tfm-web-service
  labels:
    app: tfm-web
spec:
  selector:
    app: tfm
  ports:
  - port: 60002
    targetPort: 60002
```

Figura 30: Fichero YAML del Service web.

Por último, aquí se expone un diagrama con la arquitectura desplegada del sistema con un diagrama de forma que sea más visual la explicación y se facilite su entendimiento.

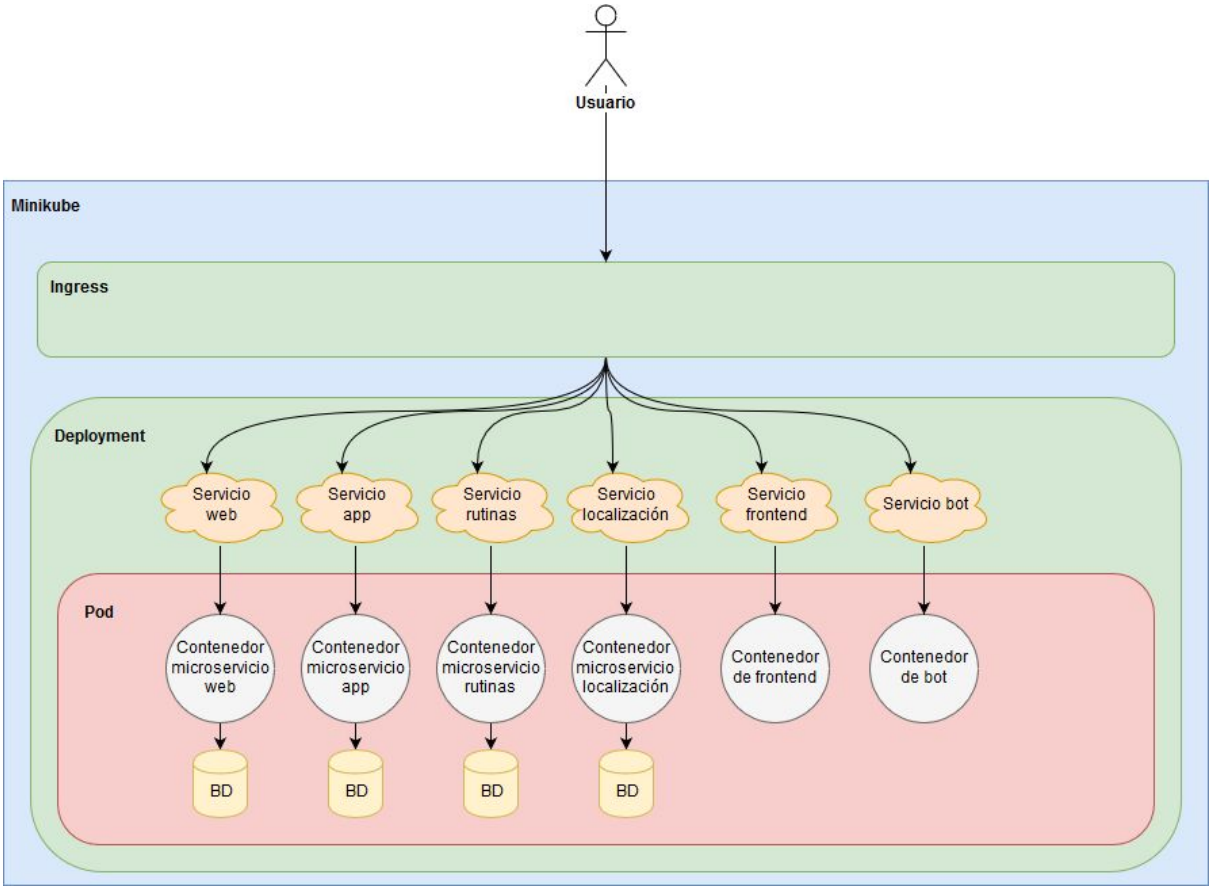

Figura 31: Diagrama de arquitectura Kubernetes.

Hay que tener en cuenta que todos los contenedores de microservicios así como las bases de datos son contenedores Docker creados a partir de imágenes desplegadas en un repositorio de DockerHub.

Despliegue final

Para probar el despliegue entero, ya que ha sido imposible en local por restricciones hardware, se ha hecho uso de la nube de Google (Gcloud) que, mediante el uso de diferentes instancias de máquinas con variadas características hardware para acomodar al máximo la arquitectura al sistema a desplegar, permite ejecutar Kubernetes para ejecutar Deployments, Services, Pods, etc.

El despliegue completo del sistema se tiene en un script bash llamado **deploy.sh** que contiene todas las intstrucciones necesarias como se puede ver en la figura de abajo.

```
#!/bin/bash

sudo kubectl create -f deployment-test.yml

sudo kubectl create -f app-service.yml
sudo kubectl create -f web-service.yml
sudo kubectl create -f routines-service.yml
sudo kubectl create -f location-service.yml
sudo kubectl create -f frontend-service.yml
sudo kubectl create -f bot-service.yml


sudo kubectl create -f db-app-service.yml
sudo kubectl create -f db-web-service.yml
sudo kubectl create -f db-routines-service.yml
sudo kubectl create -f db-location-service.yml

sudo kubectl create -f ingress-test.yaml
```

Figura 32: Instrucciones del script para realizar el despliegue.

Cuando se ejecuta dicho script, **teniendo como contexto a usar el cluster desplegado en GCloud**, se desplegará todo el sistema. Por desgracia, en la cuenta usada únicamente se pueden desplegar 5 servicios cuando en el sistema se tienen 11 por lo que ha sido imposible probarla, aún así, se puede inferir que la arquitectura hardware mínima para desplegarlo es la siguiente:

- Clúster con dos nodos, cada nodo tiene:
    - Versión → 1.9.7-gke.6
    - Imagen → Container-Optimized OS
    - Tipo de máquina → n1-standard-2
        - vCPUs → 2 *x nodo*
        - memoria RAM→ 7.5GB *x nodo*
        - Disco duro → 100 GB *x nodo*

Lo que supone un coste mensual de 78.63 € teniendo en cuenta que las máquinas estén ejecutando **18 horas (06.00 - 00.00)** mientras que si se ejecuta las 24 horas sería un coste mensual de 91.73 €.

Conclusiones

Tras haber implementado todo el sistema se ha llegado a la conclusión que una arquitectura de microservicios es muy útil en el desarrollo de una aplicación que se va a desplegar como un Backend al que se acceder como una API Rest dado que agiliza de gran manera la implementación y, además, permite exprimir al máximo las funcionalidades de Docker y Kubernetes (por ejemplo el **autoescalado**).

El uso de Docker en un sistema que tiene una arquitectura dividida en varios proyectos que necesitan conexión entre ellos (en nuestro caso los microservicios) es una solución muy importante dado que permite el despliegue de estos diferentes proyectos de una forma sencilla y rápida, únicamente creando un fichero con aproximadamente 10 líneas de código, además que permite que cualquier persona obtenga acceso a estos al permitir subirlo a un repositorio de imágenes propio de Docker. Además, al usar Kubernetes se puede tener configuración extra sobre la comunicación entre estos proyectos, como por ejemplo en la red o la persistencia de cada uno.

Inclusión en el catálogo de **Alejandría**

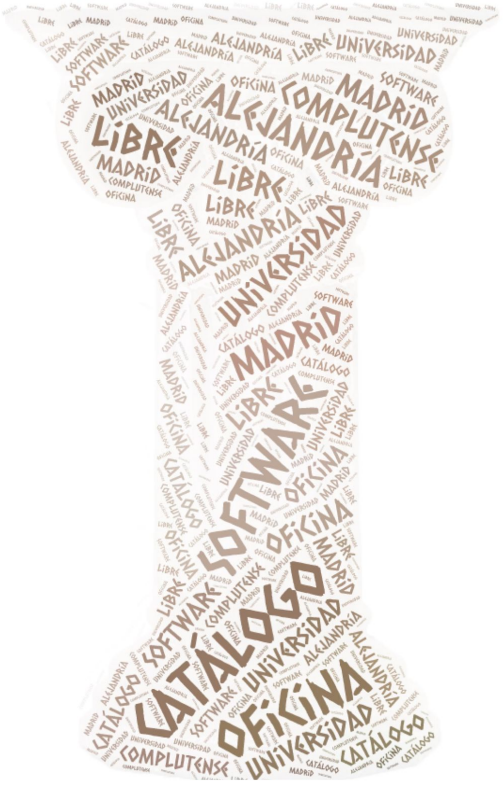

Figura 32: Logo de Alejandría.

Como uno de las últimas acciones llevadas a cabo en el proyecto se ha incluido en el catálogo de Alejandría de la universidad Complutense liberando de esta forma el código como Open Source para que cualquiera pueda acceder a este y usarlo para un sistema derivado.

¿Qué es Alejandría?

"Alejandría" es un catálogo de Trabajos Fin de Grado y Máster realizados en la Universidad Complutense de Madrid cuyo código ha sido liberado por parte de sus desarrolladores.

"Alejandría" pretende ser un escaparate más del trabajo de aquellos estudiantes que han decidido apostar por el software libre contribuyendo con su propio código, así como una muestra de la apuesta de la Universidad Complutense de Madrid por esta filosofía.

Conclusions

After implementing the complete system, the conclusion was that a microservice architecture is very useful when developing an application which is going to be deploy as a Backend accessed by an API Rest because it greatly speeds up the implementation and, besides, allows using to the fullest the Docker and Kubernetes functionalities (auto escalated for example).

Using Docker in a system with a varios subprojects structure having all of them connected is a very important solution because it allows deploying the various projects in a quick and easy way, the only thing needed is creating a ten line file, besides with this solution anyone can get access to the images given that all of the images have to be in a Docker repository. Using Kubernetes also allows the configuration in how the machines communicate and the persistence of each one of them.

Included in the **Alejandría** catalog

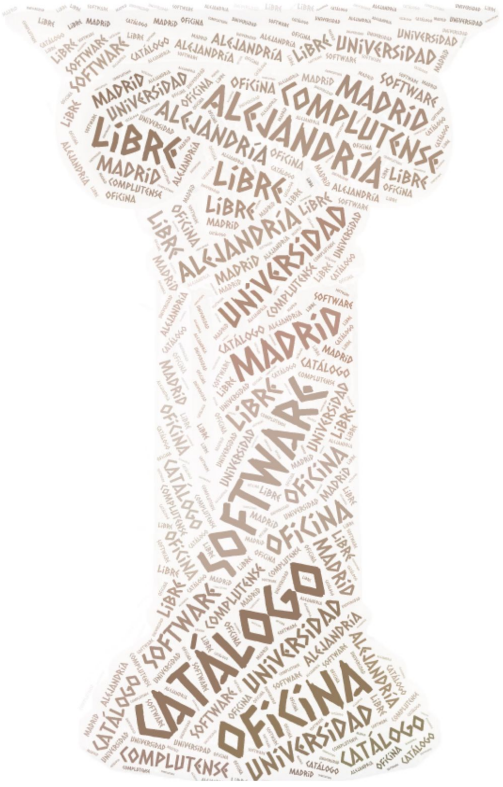

Figure 33: Alejandría logo.

As one of the last actions taken in the project, the system has been uploaded to the Alejandria of Complutense University catalog, that way liberating the code as open source so anyone can get access and use it for a derived project.

What is Alejandría?

"Alejandría" es un catálogo de Trabajos Fin de Grado y Máster realizados en la Universidad Complutense de Madrid cuyo código ha sido liberado por parte de sus desarrolladores.

“Alejandría” is a final degree and final master degree projects catalog from Complutense University of Madrid whose code has been liberated by the developers as open source.

“Alejandría” pretends to be a showcase for the students projects that are liberated as open source software, as well as showing the same philosophy from the University.

Posibles mejoras a futuro

Existen una gran cantidad de mejoras que se pueden implementar en el sistema para conseguir una mejor experiencia de cara a lo usuarios de la aplicación para que se introduzcan aún más en el mundo del juego y mejoras de cara al usuario administrativo para facilitar su trabajo y mejorar su experiencia de usuario. Algunas de las mejoras a realizar son las siguientes:

- Conectar el sistema con **Kahoot!**

  Como se ha mencionado en el estudio previo, una de las funcionalidades que nombraron los trabajadores del hospital como interesantes fue la comunicación del sistema con el juego Kahoot!, una aplicación que permite crear competiciones de responder preguntas de forma individual o en grupo.
  Esta comunicación debería ser hecha por el bot que permitiera a un administrativo crear una sala Kahoot e introducir a pacientes para que realicen las pruebas propuestas, además se podría usar también como pruebas evaluadoras o exámenes en los colegios. Esta comunicación se debería realizar por llamadas HTTP como se puede apreciar en el código del proyecto *quiztools* que está disponible en Github [https://github.com/hplgit/quiztools].

- Añadir las funcionalidades pedidas por los profesores

  A parte de la conexión con Kahoot! los profesores comentaron una serie de funcionalidades como interesantes para añadir en el sistema para facilitarle el trabajo lo máximo posible a estos y mejorar la experiencia a los alumnos. Entre las funcionalidades que destacan están las siguientes:

  - *Poder avisar al profesor por un mensaje cuántos alumnos se esperan en clase ese día.* Lo único que debería hacer el sistema es a unos minutos antes de empezar la hora de clase (por ejemplo cinco minutos antes) contar la cantidad de pacientes que tienen una tarea de asistir al colegio y enviar dicha información por el bot al administrativo marcado como profesor.

  - *Permitir evaluar por rendimiento a los alumnos y, en base a la puntuación que obtengan, otorgar recompensas en el juego.* Se podría conseguir añadiendo un flujo de conversación al bot que permitiera al profesor indicar el resultado que ha tenido el estudiante en el exámen de forma que se calcule una recompensa en base a esto.

- Mejorar el sistema de rutas con rutas en tiempo real al estilo de google maps.

  Dado que, por problemas de tiempo y de rendimiento no se ha podido dar más complejidad a la creación y visualización de rutas se deja como posible trabajo a

futuro. La idea de mejora sería que, en vez de mostrar únicamente la distancia que falta y la dirección que debe seguir se podría calcular una ruta completa y mostrarla de la misma forma que hace el servicio de Google Maps indicando todo el camino que debe seguir con pasillos que coger y habitaciones que atravesar.

Para esto primero sería necesario que en el gestor de mapas se configurara todos los obstáculos posibles del mismo (muros, puertas, mostradores, plantas, fuentes, etc) cualquier cosa que pueda suponer un estorbo a la hora de crear la ruta para el usuario, una vez estuviera esta configuración lo que debería hacer el algoritmo es calcular la ruta desde un punto A a un punto B teniendo en cuenta dichos obstáculos; teniendo en cuenta el rendimiento, la necesidad de encontrar una ruta y que dicha ruta sea la óptima, el algoritmo que se debería usar es el algoritmo de Lee, un algoritmo de coste $O(\log_n)$ que asegura encontrar la ruta siempre que exista una y que sea la óptima.

La ruta después se mostraría en la aplicación móvil usando un canvas con líneas de cada punto a punto de la misma.

- Añadir un panel de administración en el bot que permita tareas como comprobar la posición de un usuario concreto, guiar a una habitación, etc.

  Esto permitirá al usuario adminsitrativo tener cierta información importante de *feedback* en relación al comportamiento y estado de los pacientes. Se podría añadir diferentes funcionalidades que sean interesantes para el sistema, como por ejemplo que, únicamente dando el usuario del que quiere conocer el estado el bot se comunique con un endpoint desplegado en app-users para conseguir su información de estado y con otro de location para conocer su localización y se la reporte usando el bot, o que se permita al administrativo acceder al sistema de guiado introduciendo el nombre de la habitación.

- Aumentar la funcionalidad del bot, por ejemplo avisando al administrativo de cambios de ruta por parte de un paciente, comportamientos extraños, etc.

Bibliografía

Páginas de documentación

[Doc.1] **Documentación de Spring**
(https://spring.io/docs**)**

[Doc.2] **Documentación de Spring Data**
(https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)

[Doc.3] **Documentación de Ionic**
(https://ionicframework.com/docs/)

[Doc.4] **Documentación de MongoDB**
(https://docs.mongodb.com/manual/)

[Doc.5] **Documentación de CISCO CMX**
(https://www.cisco.com/c/en/us/td/docs/wireless/mse/10-3/cmx_config/b_cg_cmx103/getting_started_with_cisco_cmx.html)

[Doc.6] **Uso de coches teledirigidos para los niños de oncología en un hospital de EEUU**
(https://bit.ly/2omivS4)

[Doc.7] **Laboratorio de pruebas de CISCO**
(https://learninglabs.cisco.com/)

Papers

[Paper.1] **Erin C. Connors, Elizabeth R. Chrastil, Jaime Sánchez y Lofti B. Merabet (2014) Action video game play and transfer of navigation and spatial cognition skills in adolescents who are blind.** ***Front. Hum. Neurosci.*** (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3949101/)

[Paper.2] **Li, Binghao & Salter, James & Dempster, Andrew & Rizos, Chris. (2018). Indoor positioning techniques based on wireless LAN.**
(https://www.researchgate.net/publication/228998356_Indoor_positioning_techniques_based_on_wireless_LAN)